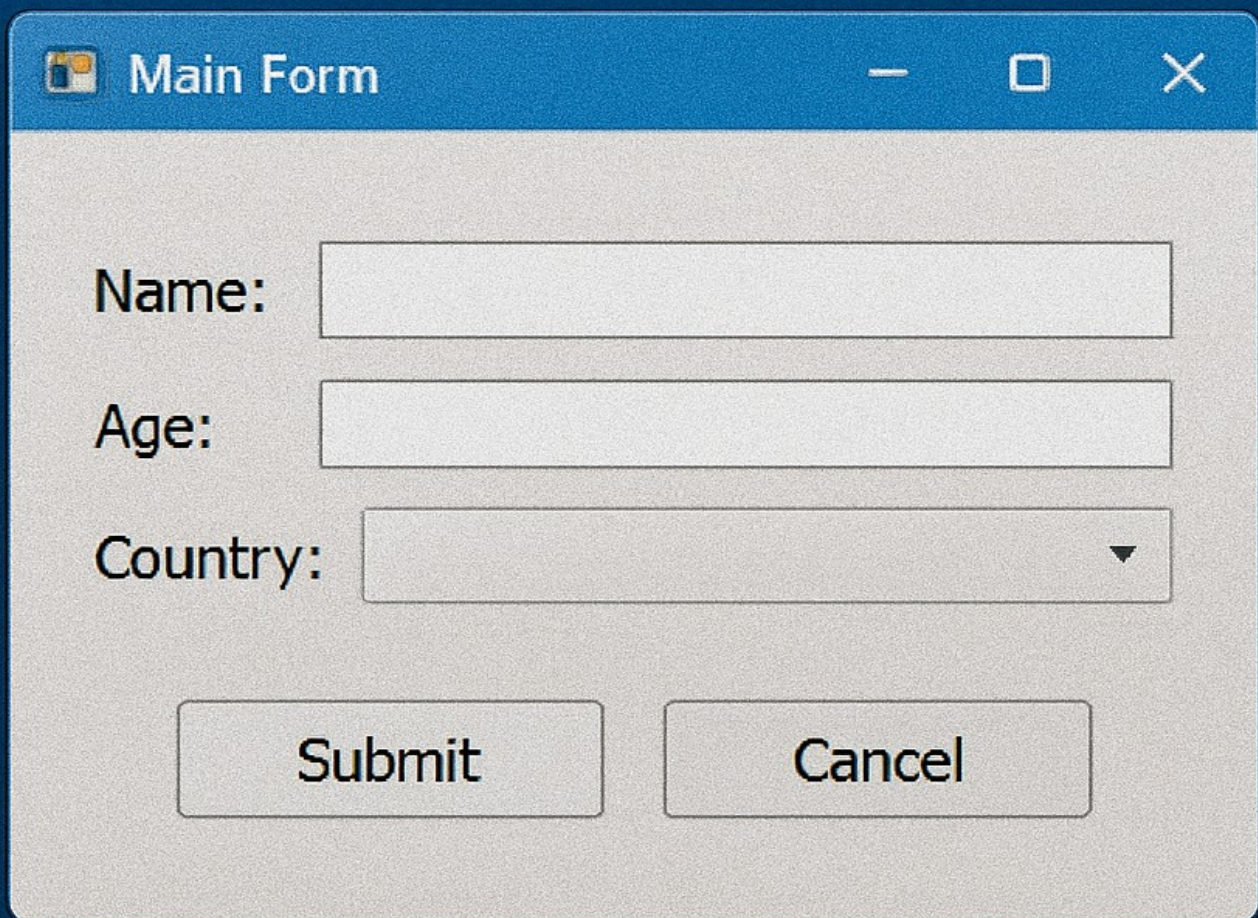


C#

WINDOWS FORM



The image shows a screenshot of a Windows Form application. The window has a blue title bar with the text "Main Form" and standard minimize, maximize, and close buttons. The form itself has a light gray background. It contains three input fields: a text box for "Name:", a text box for "Age:", and a dropdown menu for "Country:". Below these fields are two buttons: "Submit" and "Cancel".

Name:

Age:

Country:

**LUIS MINGUILLÓN
PASCUAL**

C# WINDOWS FORM

Luis Minguillón Pascual

COPYRIGHT

La licencia de este libro electrónico es para uso personal. Por lo tanto no puedes revenderlo a otras personas.

Gracias por respetar los derechos de autor.

Prefacio

Vivimos en una era donde la tecnología permea cada aspecto de nuestra vida diaria, y el desarrollo de software se ha convertido en una habilidad esencial para profesionales de múltiples áreas. Entre los muchos lenguajes de programación disponibles, C# destaca como una herramienta poderosa y versátil, diseñada por Microsoft para ofrecer un equilibrio entre eficiencia, simplicidad y robustez.

C# permite desarrollar aplicaciones de escritorio, web, móviles y juegos, pero uno de sus ámbitos más apreciados es la creación de **aplicaciones de Windows Forms**. Este tipo de aplicaciones proporciona una interfaz gráfica amigable para el usuario, permitiendo que programas complejos se presenten de manera intuitiva y profesional. Gracias a Windows Forms, los desarrolladores pueden construir programas interactivos, con controles visuales como botones, cuadros de texto, listas y paneles, sin necesidad de profundos conocimientos de bajo nivel sobre el sistema operativo.

El objetivo de este libro es guiar al lector paso a paso en el aprendizaje de C# aplicado a Windows Forms, desde los conceptos básicos hasta técnicas más avanzadas. Aquí encontrarás ejemplos claros, ejercicios prácticos y explicaciones detalladas, diseñados para que cualquier persona, desde principiantes hasta desarrolladores con experiencia, pueda comprender y aplicar los conocimientos adquiridos.

Más allá de enseñar un lenguaje, este libro busca transmitir la **importancia de C# como herramienta de creación**, mostrando cómo dominarlo puede abrir puertas a la innovación, al desarrollo de proyectos propios y al crecimiento profesional en un mercado laboral cada vez más competitivo. Aprender C# es, en definitiva, invertir en habilidades que combinan lógica, creatividad y resolución de problemas, elementos fundamentales en el mundo de la programación moderna.

Este prefacio es una invitación a adentrarse en el fascinante universo de C# y Windows Forms. Cada página de este libro está diseñada para despertar curiosidad, fomentar la práctica y consolidar conocimientos que permitirán al lector desarrollar aplicaciones funcionales, elegantes y eficientes. Bienvenido a este viaje de aprendizaje, creatividad y programación.

Luis Minguillón Pascual

INDICE:

Lección 1: Introducción a Windows Forms

- **1.1. Qué es Windows Forms y cómo funciona el modelo de eventos**
 - Windows Forms como tecnología de desarrollo para aplicaciones gráficas de escritorio en sistemas operativos Windows.
 - La abstracción sobre la API de Windows.
 - El modelo de programación orientado a eventos.
 - Definición de evento y manejador de eventos (event handler).
- **1.2. Estructura de un formulario y ciclo de vida**
 - Estructura básica: Formulario como clase que hereda de `System.Windows.Forms.Form`.
 - Ejemplo de estructura básica y uso de `Application.Run`.
 - Etapas del ciclo de vida del formulario (Creación, Carga, Activación/Visualización, Interacción, Cierre).
 - Eventos específicos del ciclo de vida (`Load`, `FormClosing`, `FormClosed`).
- **1.3. El espacio de nombres System.Windows.Forms**
 - Agrupación de clases en `System.Windows.Forms`.
 - Principales clases del espacio de nombres (`Form`, `Button`, `Label`, `TextBox`, `ListBox`, `ComboBox`, `CheckBox`, `RadioButton`, `MenuStrip`, `MessageBox`).
 - Espacios de nombres relacionados (`System.Drawing`, `System.ComponentModel`, `System.Data`).
- **1.4. Creación, diseño y ejecución de un formulario**
 - a) Creación mediante código.
 - b) Creación mediante diseñador visual (Visual Studio).
 - Ejecución del formulario y el bucle principal de mensajes.
- **1.5. Relación entre formularios y controles**
 - El formulario como contenedor de controles (jerarquía mediante la colección `Controls`).
 - Propiedades comunes heredadas de la clase `Control` (`Text`, `Location`, `Size`, `Enabled`, `Visible`, `BackColor`, `ForeColor`).
 - Comunicación entre formularios (`Show()`, `ShowDialog()`, envío/recepción de datos).

Lección 2: El Objeto Form

- **2.1. Propiedades principales del formulario**
 - `Name`, `Text`, `StartPosition` (`CenterScreen`, `WindowsDefaultLocation`, etc.), `BackColor`, `ForeColor`.
 - `Icon`, `WindowState` (`Normal`, `Maximized`, `Minimized`), `Size`.
 - `FormBorderStyle` (`None`, `FixedSingle`, `Sizable`, etc.), `ShowInTaskbar`, `ControlBox`.
- **2.2. Eventos más usados del formulario**
 - `Load`, `FormClosing`, `FormClosed`, `Resize`, `Shown`.
 - `Activated`, `Deactivate`.
 - `KeyDown`, `KeyPress`, `MouseMove`.

- **2.3. Métodos importantes del formulario**
 - Show(), Hide(), Close(), Refresh(), CenterToScreen(), BringToFront(), SendToBack().
- **2.4. Ejemplo práctico: Ventana de bienvenida con eventos de apertura y cierre**

Lección 3: Objeto Label en Windows Forms

- **3.1. Propiedades principales del Label**
 - Text, Font, ForeColor, BackColor.
 - BorderStyle (None, FixedSingle, Fixed3D), TextAlign (ContentAlignment).
 - AutoSize.
- **3.2. Eventos más usados del Label**
 - Click, MouseEnter, MouseLeave.
- **3.3. Métodos importantes del Label**
 - Refresh(), Focus().
- **3.4. Ejemplo práctico: Mensaje dinámico con cambio de color al pasar el ratón**

Lección 4: Objeto TextBox en Windows Forms

- **4.1. Propiedades principales del TextBox**
 - Text, Multiline, ReadOnly, PasswordChar, MaxLength.
 - TextAlign (HorizontalAlignment), BackColor, ForeColor.
- **4.2. Eventos más usados del TextBox**
 - TextChanged, KeyDown, KeyPress, KeyUp.
 - GotFocus, LostFocus.
- **4.3. Métodos importantes del TextBox**
 - Clear(), AppendText(string texto), SelectAll(), Copy(), Paste(), Undo().
- **4.4. Ejemplo práctico: Validación de entrada de texto y restricción de caracteres**

Lección 5: Objeto Button en Windows Forms

- **5.1. Propiedades principales del Button**
 - Text, Enabled.
 - FlatStyle (Standard, Flat, Popup, System), Image, ImageAlign.
 - BackColor, ForeColor.
- **5.2. Eventos más usados del Button**
 - Click, MouseEnter, MouseLeave, MouseDown, MouseUp.
- **5.3. Métodos importantes del Button**
 - PerformClick(), Focus(), Refresh().
- **5.4. Ejemplo práctico: Calculadora básica con varios botones**

Lección 6: Objeto CheckBox en Windows Forms

- **6.1. Propiedades principales del CheckBox**
 - Text, Checked.
 - CheckAlign (ContentAlignment), ThreeState.
- **6.2. Eventos más usados del CheckBox**
 - CheckedChanged, Click.

- **6.3. Métodos importantes del CheckBox**
 - Focus(), Refresh().
- **6.4. Ejemplo práctico: Activar o desactivar funciones con casillas**

Lección 7: Objeto RadioButton en Windows Forms

- **7.1. Propiedades principales del RadioButton**
 - Text, Checked.
 - Appearance (Normal, Button).
- **7.2. Eventos más usados del RadioButton**
 - CheckedChanged, Click.
- **7.3. Métodos importantes del RadioButton**
 - Focus(), Refresh().
- **7.4. Ejemplo práctico: Selección de opciones exclusivas**

Lección 8: Objeto ComboBox en Windows Forms

- **8.1. Propiedades principales del ComboBox**
 - Text, Items, SelectedIndex.
 - DropDownStyle (DropDown, DropDownList, Simple), Sorted.
- **8.2. Eventos más usados del ComboBox**
 - SelectedIndexChanged, TextChanged, DropDownClosed.
- **8.3. Métodos importantes del ComboBox**
 - Items.Add(object item), Items.Clear(), FindString(string s), ResetText().
- **8.4. Ejemplo práctico: Listado dinámico de opciones**

Lección 9: Objeto ListBox en Windows Forms

- **9.1. Propiedades principales del ListBox**
 - Items, SelectedIndex.
 - MultiColumn, SelectionMode (One, MultiSimple, MultiExtended).
- **9.2. Eventos más usados del ListBox**
 - SelectedIndexChanged, DoubleClick.
- **9.3. Métodos importantes del ListBox**
 - Items.Add(object item), Items.Remove(object item), ClearSelected(), FindString(string s).
- **9.4. Ejemplo práctico: Lista de tareas con eliminación dinámica**

Lección 10: Objeto PictureBox en Windows Forms

- **10.1. Propiedades principales del PictureBox**
 - Image, SizeMode (Normal, StretchImage, AutoSize, CenterImage, Zoom).
 - BorderStyle, BackColor.
- **10.2. Eventos más usados del PictureBox**
 - Click, MouseEnter, MouseLeave, MouseDown.
- **10.3. Métodos importantes del PictureBox**
 - Load(), Refresh(), Dispose().
- **10.4. Ejemplo práctico: Galería de imágenes con clic interactivo**

Lección 11: El objeto Timer en C#

- **11.1 Propiedades del Timer**
 - Interval (milisegundos).
 - Enabled (true/false).
 - **11.2 Eventos del Timer**
 - Tick.
 - **11.3 Métodos del Timer**
 - Start(), Stop(), Dispose().
 - **11.4 Ejemplos prácticos**
 - Ejemplo 1: Reloj digital.
 - Ejemplo 2: Animación automática.
-

Tercera Parte: Controles de Contenedor y Organización

Lección 12: Objeto GroupBox y Panel

- **12.1 Propiedades principales**
 - Text, BackColor, Enabled, Dock (Top, Bottom, Left, Right, Fill), Anchor.
- **12.2 Métodos importantes**
 - Refresh(), Focus(), Controls.Add().
- **12.3 Ejemplo práctico: Agrupación lógica de controles**

Lección 13: Objeto TabControl

- **13.1 Propiedades principales**
 - TabPages, SelectedIndex, Alignment (Top, Bottom, Left, Right).
- **13.2 Eventos importantes**
 - SelectedIndexChanged.
- **13.3 Métodos importantes**
 - TabPages.Add(), TabPages.Remove().
- **13.4 Ejemplo práctico: Aplicación con pestañas**

Lección 14: Objeto SplitContainer y FlowLayoutPanel

- **14.1 Propiedades principales**
 - SplitContainer (Orientation, SplitterDistance, Panel1 y Panel2).
 - FlowLayoutPanel (FlowDirection, WrapContents, AutoScroll).
 - **14.2 Eventos importantes**
 - SplitterMoved (SplitContainer).
 - **14.3 Ejemplo práctico: Diseño adaptable a redimensionamiento**
-

Cuarta Parte: Controles de Datos y Avanzados

Lección 15: Objeto DataGridView

- **15.1 Propiedades principales**
 - DataSource, Columns, Rows, ReadOnly, SelectionMode (FullRowSelect, CellSelect, etc.).
- **15.2 Eventos importantes**
 - CellClick, CellValueChanged, RowHeaderMouseClick.
- **15.3 Métodos importantes**
 - Refresh(), ClearSelection(), Rows.Add(), Rows.Remove().
- **15.4 Ejemplo práctico: Tabla editable de productos**

Lección 16. Objeto DateTimePicker y MonthCalendar

- **16.1 Propiedades principales**
 - DateTimePicker (Value, Format, MinDate, MaxDate).
 - MonthCalendar (SelectionStart y SelectionEnd, MaxSelectionCount, TodayDate).
- **16.2 Eventos importantes**
 - ValueChanged (DateTimePicker), DateChanged y DateSelected (MonthCalendar).
- **16.3 Métodos importantes**
 - Refresh(), Focus().
- **16.4 Ejemplo práctico: Selección de fechas en formularios**

Lección 17. Objeto ProgressBar

- **17.1 Propiedades principales**
 - Minimum, Maximum, Value, Step.
 - Style (Blocks, Continuous, Marquee).
- **17.2 Métodos importantes**
 - PerformStep(), Increment().
- **17.3 Ejemplo práctico: Indicador de carga de procesos**

Lección 18. Objeto TrackBar

- **18.1 Propiedades principales**
 - Minimum, Maximum, Value, TickFrequency.
- **18.2 Eventos importantes**
 - Scroll, ValueChanged.
- **18.3 Ejemplo práctico: Control de volumen o brillo**

Lección 19. Objeto ToolTip

- **19.1 Propiedades principales**
 - Active, AutomaticDelay.
 - ToolTipIcon (None, Info, Warning, Error), ToolTipTitle.
- **19.2 Métodos importantes**
 - SetToolTip(), RemoveAll().
- **19.3 Ejemplo práctico: Descripciones emergentes en botones**

Quinta Parte: Diálogos y Utilidades del Sistema

Lección 20. Objetos de Diálogo Común

- **20.1 OpenFileDialog** (Propiedades: FileName, Filter, Multiselect; Método: ShowDialog()).
 - **20.2 SaveFileDialog** (Propiedades: FileName, Filter; Método: ShowDialog()).
 - **20.3 FontDialog y ColorDialog** (Propiedades: Font, Color; Método: ShowDialog()).
 - **20.4 Ejemplo práctico: Editor de texto con cambio de fuente y color**
-

Sexta Parte: Eventos y Funciones Generales

Lección 21. Eventos Globales en Windows Forms

- **21.1. Eventos del teclado**
 - KeyDown, KeyPress, KeyUp.
 - Uso de KeyPreview = true.
- **21.2. Eventos del ratón**
 - MouseClick, MouseMove, MouseWheel.
- **21.3. Eventos de enfoque**
 - Enter, Leave, GotFocus, LostFocus.
- **21.4. Ejemplo: Interfaz interactiva con atajos de teclado**

Lección 22. Funciones Globales y de Control

- **22.1. Métodos comunes de todos los controles**
 - Focus(), Refresh(), Invalidate(), Show(), Hide(), Dispose().
 - **22.2. Manejo dinámico de controles en tiempo de ejecución**
 - Crear controles dinámicamente.
 - Eliminar controles dinámicamente (Controls.Remove(), Dispose()).
 - **22.3. Ejemplo: Formulario que genera botones dinámicos**
-

Séptima Parte: Proyectos Prácticos con Objetos y Eventos

- **23. Proyecto 1: Calculadora Científica con TextBox y Buttons**
 - **24. Proyecto 2: Agenda de Contactos con ListBox y Archivos**
 - **25. Proyecto 3: Reproductor de Imágenes con PictureBox y Timer**
 - **26. Proyecto 4: Sistema de Inventario con DataGridView y Menús**
 - **27. Proyecto 5: Editor de Texto con Diálogos de Fuente y Color**
-

Anexos de Windows Forms

- **A. Tabla completa de controles y sus eventos**

- **B. Tabla de métodos comunes en Windows Forms**
 - Focus(), Refresh(), Invalidate(), Show(), Hide(), Dispose(), BringToFront(), SendToBack(), Update(), CreateControl().
- **C. Propiedades más utilizadas por tipo de control**

Lección 1: Introducción a Windows Forms

1.1. Qué es Windows Forms y cómo funciona el modelo de eventos

Windows Forms es una tecnología de desarrollo incluida en el framework de .NET (tanto en .NET Framework clásico como en .NET Core y .NET 5/6/7 en adelante) que permite crear **aplicaciones gráficas de escritorio** para sistemas operativos Windows.

Se basa en el uso de **formularios (Forms)** como ventanas principales, que pueden contener diversos **controles gráficos** (botones, cuadros de texto, etiquetas, menús, listas, barras de progreso, etc.).

Windows Forms proporciona una capa de abstracción sobre la API de Windows, permitiendo al desarrollador centrarse en el **diseño visual y la lógica de interacción** en lugar de programar directamente con las funciones del sistema operativo.

Modelo de eventos en Windows Forms

El modelo de programación de Windows Forms es **orientado a eventos**.

Esto significa que el flujo de ejecución del programa depende de las **acciones del usuario** (como hacer clic en un botón, mover el ratón o escribir en un campo de texto) y de los **eventos del sistema** (como la carga de la ventana o su cierre).

Un **evento** es una notificación de que algo ha ocurrido.

Por ejemplo:

- Cuando el usuario hace clic en un botón, se lanza el evento `Click`.
- Cuando se cambia el texto de una caja de texto, se lanza el evento `TextChanged`.
- Cuando el formulario se abre por primera vez, se lanza el evento `Load`.

Cada evento puede estar asociado a un **manejador de eventos** (o *event handler*), que es un método que define qué debe hacer el programa cuando ese evento ocurre.

Ejemplo simple en C#:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Has hecho clic en el botón.");
}
```

En este ejemplo:

- `button1_Click` es el **manejador de evento**.
- `sender` representa el objeto que generó el evento (en este caso, el botón).
- `EventArgs e` contiene información adicional sobre el evento.

El **modelo de eventos** es la base de toda aplicación Windows Forms: el programa espera pasivamente las acciones del usuario y reacciona a ellas mediante estos manejadores.

1.2. Estructura de un formulario y ciclo de vida

Estructura básica de un formulario

Un **formulario (Form)** es la unidad principal de una aplicación Windows Forms. Es una clase que hereda de la clase base `System.Windows.Forms.Form`.

Cada formulario tiene sus propias propiedades, eventos y métodos que permiten configurarlo y controlarlo.

Ejemplo de estructura básica:

```
using System;
using System.Windows.Forms;

namespace MiAplicacion
{
    public class FormularioPrincipal : Form
    {
        public FormularioPrincipal()
        {
            this.Text = "Mi primer formulario";
            this.Width = 400;
            this.Height = 300;
        }

        [STAThread]
        static void Main()
        {
            Application.Run(new FormularioPrincipal());
        }
    }
}
```

En este código:

- Se crea una clase `FormularioPrincipal` que hereda de `Form`.
- En el constructor se definen propiedades básicas del formulario.
- En el método `Main`, se inicia la aplicación con `Application.Run`.

Ciclo de vida de un formulario

El ciclo de vida de un formulario comprende varias etapas que ocurren desde su creación hasta su cierre.

Estas etapas pueden ser controladas mediante eventos específicos.

1. Creación del formulario

- Se inicializan sus componentes (`InitializeComponent`).
- Se asignan propiedades y se crean los controles.
- Evento: `HandleCreated`.

2. Carga del formulario

- Se ejecuta el código de inicialización (como establecer valores iniciales).
- Evento: `Load`.

3. Activación y visualización

- El formulario se muestra en pantalla.
- Evento: `Shown`.

4. Interacción del usuario

- El formulario responde a eventos de controles (clics, teclas, etc.).
- Eventos típicos: `Click`, `TextChanged`, `MouseMove`, etc.

5. Cierre del formulario

- Antes de cerrarse se lanza el evento `FormClosing`, donde puede cancelarse el cierre.
- Luego se ejecuta `FormClosed` tras cerrarse completamente.

Ejemplo:

```
private void Form1_Load(object sender, EventArgs e)
{
    MessageBox.Show("El formulario se ha cargado.");
}
```

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    DialogResult respuesta = MessageBox.Show("¿Deseas salir?", "Confirmación",
    MessageBoxButtons.YesNo);
    if (respuesta == DialogResult.No)
    {
        e.Cancel = true; // Evita el cierre
    }
}
```

```
}  
}
```

1.3. El espacio de nombres `System.Windows.Forms`

Todo el conjunto de clases necesarias para desarrollar aplicaciones Windows Forms se agrupa dentro del **espacio de nombres `System.Windows.Forms`**.

Este espacio pertenece al ensamblado `System.Windows.Forms.dll`.

Principales clases del espacio de nombres

Clase	Descripción
Form	Representa una ventana o cuadro de diálogo.
Button	Control que ejecuta una acción al hacer clic.
Label	Muestra texto informativo.
TextBox	Permite introducir texto.
ListBox	Muestra una lista de elementos.
ComboBox	Muestra una lista desplegable.
CheckBox	Permite activar o desactivar una opción.
RadioButton	Permite seleccionar una sola opción entre varias.
MenuStrip	Crea menús en la parte superior del formulario.
MessageBox	Muestra cuadros de diálogo con mensajes al usuario.

Espacios relacionados

Además de `System.Windows.Forms`, existen otros espacios que complementan la interfaz:

- `System.Drawing`: para colores, fuentes, imágenes y gráficos.
- `System.ComponentModel`: para manejar componentes y propiedades.
- `System.Data`: para enlazar controles a datos (bases de datos, tablas, etc.).

1.4. Creación, diseño y ejecución de un formulario

Windows Forms puede crearse de dos maneras:

a) Creación mediante código

Puedes crear formularios y controles manualmente desde el código fuente.

Ejemplo:

```
using System;  
using System.Windows.Forms;  
  
namespace EjemploManual  
{  
    public class MiFormulario : Form
```

```

{
    public MiFormulario()
    {
        Button boton = new Button();
        boton.Text = "Saludar";
        boton.Location = new System.Drawing.Point(100, 100);
        boton.Click += (s, e) => MessageBox.Show("Hola desde Windows
Forms");

        this.Controls.Add(boton);
    }

    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new MiFormulario());
    }
}

```

En este ejemplo se crea un formulario con un botón que muestra un mensaje al hacer clic.

b) Creación mediante diseñador visual

Visual Studio ofrece un **diseñador gráfico** donde se pueden arrastrar y soltar controles sobre el formulario.

Cada control se configura desde la ventana **Propiedades** y los eventos se gestionan haciendo doble clic sobre ellos para generar los manejadores correspondientes.

Ejecución del formulario

Para ejecutar una aplicación de Windows Forms:

1. Compila el proyecto.
2. El punto de entrada debe incluir `Application.Run(new Form1());`.
3. La aplicación iniciará su **bucle principal de mensajes**, encargado de escuchar y despachar eventos.

1.5. Relación entre formularios y controles

En Windows Forms, el formulario actúa como **contenedor** de controles.

Cada control (botón, etiqueta, cuadro de texto, etc.) es un **objeto hijo** del formulario.

Esta jerarquía se organiza mediante la **colección Controls** del formulario, que almacena todos los elementos visuales que contiene.

Ejemplo:

```
Button boton1 = new Button();  
TextBox cajaTexto = new TextBox();
```

```
this.Controls.Add(boton1);  
this.Controls.Add(cajaTexto);
```

Los controles también pueden contener **otros controles** (por ejemplo, un **Panel** que agrupa varios botones).

Esto permite crear interfaces organizadas y estructuradas.

Cada control hereda de la clase base **Control**, que define propiedades comunes:

- **Text**: texto visible del control.
- **Location**: posición en el formulario.
- **Size**: tamaño del control.
- **Enabled**: habilita o deshabilita el control.
- **Visible**: muestra u oculta el control.
- **BackColor** y **ForeColor**: colores de fondo y texto.

Comunicación entre formularios

En aplicaciones más complejas, es común tener varios formularios (por ejemplo, una ventana principal y ventanas secundarias).

Un formulario puede:

- Mostrar otro con `form2.Show()`.
- Esperar su cierre con `form2.ShowDialog()`.
- Enviar o recibir datos entre ellos mediante propiedades o constructores.

Ejemplo:

```
Form2 ventanaSecundaria = new Form2();  
ventanaSecundaria.ShowDialog(); // Muestra el segundo formulario de forma modal
```

Conclusión

Windows Forms es una de las tecnologías más sólidas y fáciles de usar para crear aplicaciones de escritorio en C#.

Su modelo de eventos, estructura jerárquica de controles y entorno visual hacen posible desarrollar interfaces intuitivas y funcionales con poco código.

En lecciones posteriores se abordará cómo:

- Personalizar la apariencia del formulario.
 - Usar menús, cuadros de diálogo y controles avanzados.
 - Manejar eventos personalizados y comunicación entre formularios.
 - Conectar la interfaz a bases de datos o servicios externos.
-

Lección 2: El Objeto Form

El **objeto Form** es el elemento central de toda aplicación Windows Forms.

Representa una **ventana o interfaz gráfica** que sirve como contenedor de los demás controles visuales (botones, cuadros de texto, menús, etiquetas, etc.).

Comprender sus propiedades, eventos y métodos es fundamental para dominar el desarrollo de aplicaciones de escritorio con C#.

2.1. Propiedades principales del formulario

Un formulario hereda de la clase `System.Windows.Forms.Form` y posee numerosas propiedades que determinan su **apariencia, comportamiento y posición** en pantalla.

A continuación se detallan las más importantes.

1. Name

- Define el **nombre interno del formulario** (identificador en el código).
- Se usa para referirse al formulario desde el código fuente.
- No afecta el texto visible de la ventana.

```
this.Name = "frmPrincipal";
```

2. Text

- Es el **título visible** que aparece en la barra superior del formulario.

```
this.Text = "Aplicación de ejemplo";
```

3. StartPosition

- Controla la **posición inicial** del formulario cuando se abre.
- Puede tomar valores del enumerador `FormStartPosition`:
 - `Manual`: posición definida por el programador.
 - `CenterScreen`: centrado en la pantalla.
 - `WindowsDefaultLocation`: posición por defecto de Windows.
 - `CenterParent`: centrado respecto al formulario padre.

```
this.StartPosition = FormStartPosition.CenterScreen;
```

4. BackColor

- Define el **color de fondo** del formulario.

```
this.BackColor = System.Drawing.Color.LightBlue;
```

5. ForeColor

- Define el **color del texto** de los controles contenidos.

```
this.ForeColor = System.Drawing.Color.DarkBlue;
```

6. Icon

- Permite asignar un **icono personalizado** a la ventana (visible en la barra de título y en la barra de tareas).

```
this.Icon = new Icon("icono.ico");
```

7. WindowState

- Determina el **estado inicial de la ventana**:
 - Normal: tamaño y posición normales.
 - Maximized: ventana maximizada.
 - Minimized: ventana minimizada.

```
this.WindowState = FormWindowState.Normal;
```

8. Size

- Establece el **tamaño del formulario** en píxeles (ancho, alto).

```
this.Size = new System.Drawing.Size(800, 600);
```

9. FormBorderStyle

- Controla el **tipo de borde y marco** del formulario.
Puede tomar valores como:
 - None: sin borde (útil para pantallas completas).
 - FixedSingle: borde fijo y fino.
 - FixedDialog: borde de cuadro de diálogo.
 - Sizable: borde redimensionable (por defecto).
 - FixedToolWindow y SizableToolWindow: ventanas tipo herramienta.

```
this.FormBorderStyle = FormBorderStyle.FixedSingle;
```

10. ShowInTaskbar

- Indica si el formulario debe **aparecer o no en la barra de tareas**.

```
this.ShowInTaskbar = true;
```

11. ControlBox

- Permite mostrar u ocultar los botones **Minimizar, Maximizar y Cerrar** de la esquina superior derecha.

```
this.ControlBox = true;
```

2.2. Eventos más usados del formulario

Los **eventos del formulario** permiten reaccionar a las acciones del usuario o del sistema operativo. A continuación se describen los más comunes y su propósito.

1. Load

Se produce **cuando el formulario se carga por primera vez**, antes de mostrarse en pantalla. Ideal para inicializar variables o configurar controles.

```
private void Form1_Load(object sender, EventArgs e)
{
    MessageBox.Show("El formulario se ha cargado correctamente.");
}
```

2. FormClosing

Ocurre **justo antes de que el formulario se cierre**.

Permite cancelar el cierre si es necesario.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    DialogResult respuesta = MessageBox.Show("¿Deseas salir?", "Confirmar salida", MessageBoxButtons.YesNo);
    if (respuesta == DialogResult.No)
        e.Cancel = true;
}
```

3. FormClosed

Se ejecuta **después de cerrarse el formulario**.

Se utiliza para liberar recursos o guardar configuraciones.

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    MessageBox.Show("El formulario ha sido cerrado.");
}
```

4. Resize

Se dispara **cuando el tamaño del formulario cambia**.

```
private void Form1_Resize(object sender, EventArgs e)
{
    this.Text = $"Tamaño actual: {this.Width}x{this.Height}";
}
```

5. Shown

Se produce **cuando el formulario ya es visible** en pantalla.

Útil para mostrar mensajes de bienvenida o iniciar animaciones.

```
private void Form1_Shown(object sender, EventArgs e)
{
}
```

```
    MessageBox.Show("Bienvenido a la aplicación");  
}
```

6. Activated

Se dispara **cuando el formulario obtiene el foco** (se convierte en la ventana activa).

```
private void Form1_Activated(object sender, EventArgs e)  
{  
    this.BackColor = Color.LightYellow;  
}
```

7. Deactivate

Se produce **cuando el formulario pierde el foco**.

```
private void Form1_Deactivate(object sender, EventArgs e)  
{  
    this.BackColor = Color.LightGray;  
}
```

8. KeyDown

Se lanza **cuando el usuario presiona una tecla del teclado**.

Permite detectar combinaciones de teclas.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)  
{  
    if (e.KeyCode == Keys.Escape)  
        this.Close();  
}
```

Para que funcione, la propiedad `KeyPreview` del formulario debe estar en `true`.

9. KeyPress

Detecta **caracteres introducidos** (letras, números, símbolos).

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)  
{  
    MessageBox.Show("Has presionado: " + e.KeyChar);  
}
```

10. MouseMove

Se activa **al mover el cursor dentro del formulario**.

```
private void Form1_MouseMove(object sender, MouseEventArgs e)  
{  
    this.Text = $"Posición del ratón: X={e.X}, Y={e.Y}";  
}
```

}

2.3. Métodos importantes del formulario

El objeto Form dispone de métodos que controlan su **comportamiento en tiempo de ejecución**. Los más usados son los siguientes:

1. Show()

Muestra el formulario de manera **no modal**, permitiendo seguir usando otros formularios.

```
Form2 ventana = new Form2();  
ventana.Show();
```

2. Hide()

Oculto temporalmente el formulario sin destruirlo.

```
this.Hide();
```

3. Close()

Cierra el formulario y libera sus recursos.

```
this.Close();
```

4. Refresh()

Fuerza la **actualización visual** del formulario y sus controles.

```
this.Refresh();
```

5. CenterToScreen()

Centra el formulario en el medio de la pantalla.

```
this.CenterToScreen();
```

6. BringToFront()

Trae el formulario al frente de todas las ventanas.

```
this.BringToFront();
```

7. SendToBack()

Envía el formulario al fondo (detrás de otros formularios o controles).

```
this.SendToBack();
```

2.4. Ejemplo práctico: Ventana de bienvenida con eventos de apertura y cierre

A continuación, se muestra un ejemplo completo que utiliza las propiedades, eventos y métodos más relevantes del formulario.

Objetivo

Crear una aplicación con una **ventana de bienvenida**, que:

- Cambie de color cuando se active o desactive.
- Muestre un mensaje al cargarse y al cerrarse.
- Se centre en pantalla.
- Permita cerrarse con la tecla ESC.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace VentanaBienvenida
{
    public class FormBienvenida : Form
    {
        public FormBienvenida()
        {
            // Propiedades principales del formulario
            this.Name = "frmBienvenida";
            this.Text = "Ventana de Bienvenida";
            this.StartPosition = FormStartPosition.CenterScreen;
            this.BackColor = Color.LightBlue;
            this.ForeColor = Color.DarkBlue;
            this.Size = new Size(500, 300);
            this.FormBorderStyle = FormBorderStyle.FixedDialog;
            this.KeyPreview = true;

            // Evento de teclado para cerrar con ESC
            this.KeyDown += FormBienvenida_KeyDown;
            // Eventos de ciclo de vida
            this.Load += FormBienvenida_Load;
            this.Shown += FormBienvenida_Shown;
            this.FormClosing += FormBienvenida_FormClosing;
            this.FormClosed += FormBienvenida_FormClosed;
            this.Activated += FormBienvenida_Activated;
```

```

        this.Deactivate += FormBienvenida_Deactivate;
    }

    private void FormBienvenida_Load(object sender, EventArgs e)
    {
        this.CenterToScreen();
    }

    private void FormBienvenida_Shown(object sender, EventArgs e)
    {
        MessageBox.Show("Bienvenido a la aplicación", "Mensaje de bienvenida");
    }

    private void FormBienvenida_FormClosing(object sender, FormClosingEventArgs e)
    {
        DialogResult resultado = MessageBox.Show("¿Deseas salir?", "Confirmación", MessageBoxButtons.YesNo);
        if (resultado == DialogResult.No)
            e.Cancel = true;
    }

    private void FormBienvenida_FormClosed(object sender, FormClosedEventArgs e)
    {
        MessageBox.Show("Gracias por usar la aplicación.");
    }

    private void FormBienvenida_Activated(object sender, EventArgs e)
    {
        this.BackColor = Color.LightGreen;
    }

    private void FormBienvenida_Deactivate(object sender, EventArgs e)
    {
        this.BackColor = Color.LightGray;
    }

```

```

private void FormBienvenida_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
        this.Close();
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormBienvenida());
}
}

```

Explicación paso a paso

1. **Configuración inicial:** se establecen propiedades como el tamaño, color y posición del formulario.
2. **Eventos:**
 - Load → prepara la posición inicial.
 - Shown → muestra el mensaje de bienvenida.
 - FormClosing → pide confirmación antes de cerrar.
 - FormClosed → muestra un mensaje final.
 - Activated y Deactivate → cambian el color de fondo según el foco.
 - KeyDown → permite cerrar la ventana con la tecla ESC.
3. **Métodos usados:**
 - CenterToScreen() para centrar.
 - Close() para finalizar la aplicación.

Este ejemplo demuestra cómo combinar **propiedades, eventos y métodos del objeto Form** para construir una ventana profesional y funcional.

Lección 3: Objeto Label en Windows Forms

El **Label** es un control gráfico de Windows Forms que se utiliza para **mostrar texto estático o dinámico** en un formulario. No permite la edición por parte del usuario, pero puede responder a eventos del ratón o del teclado para crear efectos visuales o interacción limitada.

El control Label es muy útil para mostrar títulos, descripciones, instrucciones o resultados dentro de la interfaz.

3.1. Propiedades principales del Label

El objeto Label posee varias propiedades que determinan su **contenido, apariencia y alineación**. Las más importantes son:

1. Text

- Contiene el **texto que se mostrará en el Label**.
- Puede actualizarse en tiempo de ejecución para mostrar mensajes dinámicos.

```
label1.Text = "Bienvenido a la aplicación";
```

2. Font

- Define la **tipografía del texto**, incluyendo tipo de letra, tamaño y estilo (negrita, cursiva, subrayado).

```
label1.Font = new Font("Arial", 14, FontStyle.Bold);
```

3. ForeColor

- Establece el **color del texto**.

```
label1.ForeColor = Color.DarkBlue;
```

4. BackColor

- Determina el **color de fondo** del Label.
- Si se desea que sea transparente, se puede usar `Color.Transparent`.

```
label1.BackColor = Color.LightGray;
```

5. BorderStyle

- Controla el **borde del Label**.
- Puede tomar valores del enumerador `BorderStyle`:
 - `None`: sin borde (por defecto).
 - `FixedSingle`: borde fino y rectangular.
 - `Fixed3D`: borde con efecto 3D.

```
label1.BorderStyle = BorderStyle.FixedSingle;
```

6. TextAlign

- Determina la **alineación del texto dentro del Label**.
- Valores del enumerador ContentAlignment:
 - TopLeft, TopCenter, TopRight
 - MiddleLeft, MiddleCenter, MiddleRight
 - BottomLeft, BottomCenter, BottomRight

```
label1.TextAlign = ContentAlignment.MiddleCenter;
```

7. AutoSize

- Indica si el Label **ajustará automáticamente su tamaño** al contenido del texto.
- Si es false, se puede establecer un tamaño fijo.

```
label1.AutoSize = true;
```

3.2. Eventos más usados del Label

Aunque el Label no permite interacción directa como un botón, puede **responder a eventos del ratón y del teclado**.

1. Click

- Se activa cuando el usuario **hace clic sobre el Label**.

```
private void label1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Has hecho clic en el mensaje.");
}
```

2. MouseEnter

- Se dispara **cuando el cursor entra en el área del Label**.
- Ideal para efectos visuales como resaltar el texto.

```
private void label1_MouseEnter(object sender, EventArgs e)
{
    label1.ForeColor = Color.Red;
}
```

3. MouseLeave

- Se ejecuta **cuando el cursor sale del área del Label**.
- Permite restaurar la apariencia original.

```
private void label1_MouseLeave(object sender, EventArgs e)
{
}
```

```
        label1.ForeColor = Color.Black;
    }
```

3.3. Métodos importantes del Label

Aunque no es un control interactivo principal, el Label posee algunos métodos útiles:

1. Refresh()

- Fuerza la **actualización inmediata del control**, útil si se cambian propiedades dinámicamente y se quiere que se reflejen al instante.

```
label1.Refresh();
```

2. Focus()

- Permite **darle el foco al Label**, aunque generalmente se usa en controles que pueden recibir entrada de datos.
- Puede ser útil para eventos de teclado asociados al Label.

```
label1.Focus();
```

3.4. Ejemplo práctico: Mensaje dinámico con cambio de color al pasar el ratón

Objetivo

Crear un formulario con un Label que:

1. Muestra un **mensaje de bienvenida**.
2. Cambia el **color del texto** cuando el ratón pasa sobre él.
3. Restaura el color original al salir el ratón.
4. Muestra un mensaje al hacer clic.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace LabelDinamico
{
    public class FormLabel : Form
    {
        private Label labelMensaje;
```

```

public FormLabel()
{
    // Configuración del formulario
    this.Text = "Ejemplo Label Dinámico";
    this.Size = new Size(400, 200);
    this.StartPosition = FormStartPosition.CenterScreen;
    this.BackColor = Color.LightYellow;

    // Creación del Label
    labelMensaje = new Label();
    labelMensaje.Text = "Bienvenido a la aplicación";
    labelMensaje.Font = new Font("Arial", 14, FontStyle.Bold);
    labelMensaje.ForeColor = Color.Black;
    labelMensaje.BackColor = Color.LightGray;
    labelMensaje.BorderStyle = BorderStyle.FixedSingle;
    labelMensaje.TextAlign = ContentAlignment.MiddleCenter;
    labelMensaje.AutoSize = false;
    labelMensaje.Size = new Size(300, 50);
    labelMensaje.Location = new Point(50, 50);

    // Eventos del Label
    labelMensaje.Click += LabelMensaje_Click;
    labelMensaje.MouseEnter += LabelMensaje_MouseEnter;
    labelMensaje.MouseLeave += LabelMensaje_MouseLeave;

    // Agregar Label al formulario
    this.Controls.Add(labelMensaje);
}

private void LabelMensaje_Click(object sender, EventArgs e)
{
    MessageBox.Show("¡Has hecho clic en el mensaje!");
}

private void LabelMensaje_MouseEnter(object sender, EventArgs e)
{

```

```

        labelMensaje.ForeColor = Color.Red;
    }

    private void LabelMensaje_MouseLeave(object sender, EventArgs e)
    {
        labelMensaje.ForeColor = Color.Black;
    }

    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new FormLabel());
    }
}

```

Explicación del ejemplo

1. **Formulario:** tamaño 400x200, fondo amarillo, centrado en pantalla.
2. **Label:**
 - Texto centrado y con borde 3D.
 - Color de texto negro por defecto, fondo gris.
 - Tamaño fijo (AutoSize = false) para permitir alineación precisa.
3. **Eventos:**
 - `MouseEnter` → cambia el color del texto a rojo.
 - `MouseLeave` → restaura el color a negro.
 - `Click` → muestra un mensaje emergente.
4. **Métodos utilizados:** `Controls.Add` para agregar el Label al formulario.

Este ejemplo demuestra cómo un **Label puede ser dinámico** y reaccionar a la interacción del usuario, a pesar de ser un control generalmente estático.

Lección 4: Objeto TextBox en Windows Forms

El **TextBox** es uno de los controles más utilizados en Windows Forms, ya que permite **ingresar y mostrar texto editable** por el usuario. Se puede configurar para aceptar **una sola línea o varias líneas**, ocultar caracteres (para contraseñas), y validar la entrada en tiempo real.

Comprender sus propiedades, eventos y métodos es esencial para crear interfaces interactivas y seguras.

4.1. Propiedades principales del TextBox

El TextBox posee numerosas propiedades que controlan su **contenido, apariencia y comportamiento**. Las más importantes son:

1. Text

- Contiene el **texto actual** dentro del TextBox.
- Puede leerse y modificarse en tiempo de ejecución.

```
textBox1.Text = "Escribe tu nombre";
```

2. Multiline

- Indica si el TextBox permite **varias líneas de texto**.
- Cuando es `true`, se pueden usar saltos de línea (`\r\n`).

```
textBox1.Multiline = true;  
textBox1.Size = new Size(300, 100);
```

3. ReadOnly

- Si se establece en `true`, el usuario **no puede modificar** el contenido, pero aún se puede copiar texto.

```
textBox1.ReadOnly = true;
```

4. PasswordChar

- Oculta los caracteres ingresados, útil para contraseñas.
- Se puede asignar un carácter como `*` o `●`.

```
textBox1.PasswordChar = '*';
```

5. MaxLength

- Limita la **cantidad máxima de caracteres** que se pueden ingresar.

```
textBox1.MaxLength = 10;
```

6. TextAlign

- Alinea el texto dentro del TextBox:

- `HorizontalAlignment.Left` (por defecto)
- `HorizontalAlignment.Center`
- `HorizontalAlignment.Right`

```
textBox1.TextAlign = HorizontalAlignment.Center;
```

7. BackColor

- Color de **fondo del TextBox**.

```
textBox1.BackColor = Color.LightYellow;
```

8. ForeColor

- Color del **texto**.

```
textBox1.ForeColor = Color.DarkBlue;
```

4.2. Eventos más usados del TextBox

El TextBox permite **interacción directa con el teclado** y responde a varios eventos:

1. TextChanged

- Se dispara **cada vez que el texto cambia**, ya sea por teclado o código.
- Ideal para validaciones en tiempo real.

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
    label1.Text = "Longitud del texto: " + textBox1.Text.Length;
}
```

2. KeyDown

- Se produce **cuando una tecla se presiona**, antes de que el carácter aparezca.
- Permite detectar combinaciones de teclas.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
        MessageBox.Show("Has presionado Enter");
}
```

3. KeyPress

- Se dispara **al presionar una tecla que genera un carácter**.
- Permite filtrar entrada de caracteres.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{

```

```
        if (!char.IsLetter(e.KeyChar) && !char.IsControl(e.KeyChar))
            e.Handled = true; // Bloquea números y símbolos
    }
```

4. KeyUp

- Ocurre **al soltar la tecla**, útil para acciones posteriores.

```
private void textBox1_KeyUp(object sender, KeyEventArgs e)
{
    // Actualiza información tras soltar la tecla
}
```

5. GotFocus

- Se activa **cuando el TextBox recibe el foco**.

```
private void textBox1_GotFocus(object sender, EventArgs e)
{
    textBox1.BackColor = Color.LightCyan;
}
```

6. LostFocus

- Se produce **cuando el TextBox pierde el foco**.

```
private void textBox1_LostFocus(object sender, EventArgs e)
{
    textBox1.BackColor = Color.White;
}
```

4.3. Métodos importantes del TextBox

El TextBox posee métodos que permiten **manipular y controlar el texto** de manera eficiente.

Método	Descripción	Ejemplo
Clear()	Borra todo el contenido del TextBox	textBox1.Clear();
AppendText(string texto)	Añade texto al final del existente	textBox1.AppendText("Hola");
SelectAll()	Selecciona todo el texto	textBox1.SelectAll();
Copy()	Copia el texto seleccionado al portapapeles	textBox1.Copy();
Paste()	Pega el contenido del portapapeles	textBox1.Paste();
Undo()	Deshace la última acción de edición	textBox1.Undo();

4.4. Ejemplo práctico: Validación de entrada de texto y restricción de caracteres

Objetivo

Crear un formulario con un TextBox que:

1. Permita ingresar solo **letras mayúsculas y minúsculas**.
2. Limite la entrada a **máximo 10 caracteres**.
3. Cambie el color de fondo cuando reciba o pierda el foco.
4. Muestre la longitud del texto en un Label en tiempo real.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace TextBoxEjemplo
{
    public class FormTextBox : Form
    {
        private TextBox txtNombre;
        private Label lblInfo;

        public FormTextBox()
        {
            // Configuración del formulario
            this.Text = "Validación TextBox";
            this.Size = new Size(400, 200);
            this.StartPosition = FormStartPosition.CenterScreen;
            this.BackColor = Color.LightYellow;

            // Configuración del TextBox
            txtNombre = new TextBox();
            txtNombre.Location = new Point(50, 50);
            txtNombre.Size = new Size(200, 30);
            txtNombre.MaxLength = 10;
            txtNombre.BackColor = Color.White;
        }
    }
}
```

```

txtNombre.ForeColor = Color.Black;
txtNombre.TextAlign = HorizontalAlignment.Left;

// Eventos del TextBox
txtNombre.KeyPress += TxtNombre_KeyPress;
txtNombre.TextChanged += TxtNombre_TextChanged;
txtNombre.GotFocus += TxtNombre_GotFocus;
txtNombre.LostFocus += TxtNombre_LostFocus;

// Label informativo
lblInfo = new Label();
lblInfo.Location = new Point(50, 100);
lblInfo.Size = new Size(300, 30);
lblInfo.Text = "Longitud del texto: 0";

// Agregar controles al formulario
this.Controls.Add(txtNombre);
this.Controls.Add(lblInfo);
}

// Permitir solo letras
private void TxtNombre_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!char.IsLetter(e.KeyChar) && !char.IsControl(e.KeyChar))
    {
        e.Handled = true;
    }
}

// Mostrar longitud del texto
private void TxtNombre_TextChanged(object sender, EventArgs e)
{
    lblInfo.Text = $"Longitud del texto: {txtNombre.Text.Length}";
}

// Cambio de color al recibir el foco

```

```

private void TxtNombre_GotFocus(object sender, EventArgs e)
{
    txtNombre.BackColor = Color.LightCyan;
}

// Restaurar color al perder el foco
private void TxtNombre_LostFocus(object sender, EventArgs e)
{
    txtNombre.BackColor = Color.White;
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormTextBox());
}
}

```

Explicación del ejemplo

1. **Formulario:** tamaño 400x200, fondo amarillo, centrado en pantalla.
2. **TextBox:**
 - Solo permite letras.
 - Máximo 10 caracteres.
 - Cambia color al recibir o perder el foco.
3. **Label:** muestra la longitud del texto en tiempo real.
4. **Eventos:**
 - KeyPress → valida entrada de caracteres.
 - TextChanged → actualiza el Label.
 - GotFocus / LostFocus → efecto visual.

Este ejemplo muestra cómo un **TextBox puede ser controlado y validado** de manera sencilla, evitando errores de entrada y mejorando la experiencia del usuario.

Lección 5: Objeto Button en Windows Forms

El **Button** es un control fundamental en Windows Forms, ya que permite **ejecutar acciones** cuando el usuario hace clic sobre él. Los botones son ideales para disparar eventos, ejecutar operaciones, abrir formularios o interactuar con otros controles como TextBox, Label o ComboBox.

5.1. Propiedades principales del Button

El Button posee propiedades que determinan su **apariciencia, estado y comportamiento**:

1. Text

- Contiene el **texto visible** en el botón.

```
button1.Text = "Aceptar";
```

2. Enabled

- Determina si el botón está **habilitado o deshabilitado**.
- Si es `false`, el botón se ve atenuado y no puede clicarse.

```
button1.Enabled = true;
```

3. FlatStyle

- Controla el **estilo visual** del botón:
 - `Standard` → estilo clásico de Windows (por defecto).
 - `Flat` → borde plano.
 - `Popup` → borde que aparece al pasar el mouse.
 - `System` → estilo definido por el sistema operativo.

```
button1.FlatStyle = FlatStyle.Flat;
```

4. Image

- Permite agregar **una imagen dentro del botón**, junto al texto.

```
button1.Image = Image.FromFile("icono.png");  
button1.ImageAlign = ContentAlignment.MiddleLeft;
```

5. BackColor

- Color de fondo del botón.

```
button1.BackColor = Color.LightBlue;
```

6. ForeColor

- Color del texto del botón.

```
button1.ForeColor = Color.DarkBlue;
```

5.2. Eventos más usados del Button

Los botones responden principalmente a **eventos del mouse**, permitiendo interacción visual y funcional:

1. Click

- Se dispara **cuando el usuario hace clic sobre el botón**.
- Es el evento más común para ejecutar código.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Has presionado el botón");
}
```

2. MouseEnter

- Se activa **cuando el cursor entra en el área del botón**, útil para efectos visuales.

```
private void button1_MouseEnter(object sender, EventArgs e)
{
    button1.BackColor = Color.LightGreen;
}
```

3. MouseLeave

- Se produce **cuando el cursor sale del botón**, para restaurar apariencia.

```
private void button1_MouseLeave(object sender, EventArgs e)
{
    button1.BackColor = Color.LightBlue;
}
```

4. MouseDown

- Se dispara **al presionar un botón del ratón sobre el control**, antes de soltarlo.

```
private void button1_MouseDown(object sender, MouseEventArgs e)
{
    button1.BackColor = Color.Green;
}
```

5. MouseUp

- Se produce **al soltar el botón del ratón** sobre el control.

```
private void button1_MouseUp(object sender, MouseEventArgs e)
{
    button1.BackColor = Color.LightBlue;
}
```

5.3. Métodos importantes del Button

Los métodos del Button permiten **simular interacciones y controlar visualmente el botón**:

1. PerformClick()

- Simula un **clic programático** sobre el botón, disparando su evento Click.

```
button1.PerformClick();
```

2. Focus()

- Asigna el **foco al botón**, permitiendo que se active con la tecla Enter si está configurado.

```
button1.Focus();
```

3. Refresh()

- Fuerza la **actualización visual del botón**.

```
button1.Refresh();
```

5.4. Ejemplo práctico: Calculadora básica con varios botones

Objetivo

Crear un **formulario con botones** que permita realizar operaciones básicas (suma, resta, multiplicación, división) sobre dos números ingresados en TextBox y mostrar el resultado en un Label.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CalculadoraBasica
{
    public class FormCalculadora : Form
    {
        private TextBox txtNum1, txtNum2;
        private Label lblResultado;
        private Button btnSumar, btnRestar, btnMultiplicar, btnDividir;

        public FormCalculadora()
        {
            // Configuración del formulario
            this.Text = "Calculadora Básica";
        }
    }
}
```

```

        this.Size = new Size(400, 300);
        this.StartPosition = FormStartPosition.CenterScreen;

        // TextBoxes
        txtNum1 = new TextBox() { Location = new Point(50, 30), Width =
100 };
        txtNum2 = new TextBox() { Location = new Point(200, 30), Width = 100
};

        // Label
        lblResultado = new Label()
        {
            Text = "Resultado:",
            Location = new Point(50, 70),
            Size = new Size(300, 30),
            Font = new Font("Arial", 12, FontStyle.Bold)
        };

        // Botones
        btnSumar = CrearBoton("Sumar", new Point(50, 120), BtnSumar_Click);
        btnRestar = CrearBoton("Restar", new Point(150, 120),
BtnRestar_Click);
        btnMultiplicar = CrearBoton("Multiplicar", new Point(250, 120),
BtnMultiplicar_Click);
        btnDividir = CrearBoton("Dividir", new Point(50, 170),
BtnDividir_Click);

        // Agregar controles al formulario
        this.Controls.Add(txtNum1);
        this.Controls.Add(txtNum2);
        this.Controls.Add(lblResultado);
        this.Controls.Add(btnSumar);
        this.Controls.Add(btnRestar);
        this.Controls.Add(btnMultiplicar);
        this.Controls.Add(btnDividir);
    }

    // Método auxiliar para crear botones

```

```

        private Button CrearBoton(string texto, Point ubicacion, EventHandler
eventoClick)
        {
            Button btn = new Button();
            btn.Text = texto;
            btn.Size = new Size(80, 30);
            btn.Location = ubicacion;
            btn.BackColor = Color.LightBlue;
            btn.ForeColor = Color.Black;
            btn.Click += eventoClick;
            btn.MouseEnter += (s, e) => btn.BackColor = Color.LightGreen;
            btn.MouseLeave += (s, e) => btn.BackColor = Color.LightBlue;
            return btn;
        }

// Eventos Click
private void BtnSumar_Click(object sender, EventArgs e)
{
    double a = Convert.ToDouble(txtNum1.Text);
    double b = Convert.ToDouble(txtNum2.Text);
    lblResultado.Text = "Resultado: " + (a + b);
}

private void BtnRestar_Click(object sender, EventArgs e)
{
    double a = Convert.ToDouble(txtNum1.Text);
    double b = Convert.ToDouble(txtNum2.Text);
    lblResultado.Text = "Resultado: " + (a - b);
}

private void BtnMultiplicar_Click(object sender, EventArgs e)
{
    double a = Convert.ToDouble(txtNum1.Text);
    double b = Convert.ToDouble(txtNum2.Text);
    lblResultado.Text = "Resultado: " + (a * b);
}

```

```

private void BtnDividir_Click(object sender, EventArgs e)
{
    double a = Convert.ToDouble(txtNum1.Text);
    double b = Convert.ToDouble(txtNum2.Text);
    if (b != 0)
        lblResultado.Text = "Resultado: " + (a / b);
    else
        MessageBox.Show("No se puede dividir entre cero", "Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormCalculadora());
}
}

```

Explicación del ejemplo

1. **Formulario:** tamaño 400x300, centrado en pantalla.
2. **TextBox:** dos controles para ingresar los números.
3. **Label:** muestra el resultado de las operaciones.
4. **Botones:**
 - Cada botón tiene un evento `Click` para ejecutar la operación correspondiente.
 - `MouseEnter` y `MouseLeave` cambian el color de fondo al pasar el ratón.
 - `CrearBoton` es un método auxiliar para reducir código repetitivo.
5. **Validación básica:** el botón Dividir verifica que no se divida entre cero y muestra un mensaje de error si ocurre.

Este ejemplo muestra cómo **varios botones pueden interactuar con TextBox y Label**, creando una calculadora funcional y demostrando el uso práctico de las propiedades, eventos y métodos del control Button.

Lección 6: Objeto CheckBox en Windows Forms

El **CheckBox** es un control que permite al usuario **seleccionar o deseleccionar opciones** dentro de un formulario. A diferencia de los botones, los CheckBox pueden permanecer en estado marcado o desmarcado, y en ciertos casos pueden tener un **tercer estado** (indeterminado).

Se utiliza frecuentemente para **activar o desactivar funciones**, configurar opciones o filtrar datos en aplicaciones de escritorio.

6.1. Propiedades principales del CheckBox

El CheckBox tiene propiedades que controlan su **estado, apariencia y alineación**:

1. Text

- Define el **texto visible junto a la casilla**, describiendo la opción.

```
checkBox1.Text = "Activar notificaciones";
```

2. Checked

- Indica si el CheckBox está **marcado (true) o desmarcado (false)**.

```
checkBox1.Checked = true; // Marcado por defecto
```

3. CheckAlign

- Determina la **posición de la casilla** respecto al texto.
- Puede usar **ContentAlignment**:
 - MiddleLeft** → casilla a la izquierda del texto (por defecto).
 - MiddleRight** → casilla a la derecha del texto.

```
checkBox1.CheckAlign = ContentAlignment.MiddleLeft;
```

4. ThreeState

- Permite que el CheckBox tenga **tres estados**:
 - false** → solo marcado o desmarcado.
 - true** → marcado, desmarcado e indeterminado (**CheckState.Indeterminate**).

```
checkBox1.ThreeState = true;
```

6.2. Eventos más usados del CheckBox

Los CheckBox responden principalmente a cambios de estado y clics:

1. CheckedChanged

- Se dispara **cada vez que cambia el estado del CheckBox**, ya sea por usuario o por código.
- Se utiliza para **activar o desactivar funcionalidades** automáticamente.

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    if (checkBox1.Checked)
        MessageBox.Show("La opción ha sido activada.");
    else
        MessageBox.Show("La opción ha sido desactivada.");
}
```

2. Click

- Se produce **cuando el usuario hace clic en la casilla**, independientemente de si cambia el estado o no.

```
private void checkBox1_Click(object sender, EventArgs e)
{
    Console.WriteLine("Se hizo clic en la casilla");
}
```

6.3. Métodos importantes del CheckBox

Aunque el CheckBox es sencillo, posee métodos útiles:

1. Focus()

- Asigna el **foco al CheckBox**, permitiendo que se active con la barra espaciadora.

```
checkBox1.Focus();
```

2. Refresh()

- Fuerza la **actualización visual** del CheckBox después de cambiar propiedades en tiempo de ejecución.

```
checkBox1.Refresh();
```

6.4. Ejemplo práctico: Activar o desactivar funciones con casillas

Objetivo

Crear un formulario con **tres CheckBox** que permitan activar o desactivar funciones de la aplicación.

- Al marcar cada casilla, se mostrará un mensaje indicando que la función está activada.

- Al desmarcarla, la función se desactiva.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CheckBoxEjemplo
{
    public class FormCheckBox : Form
    {
        private CheckBox chkNotificaciones;
        private CheckBox chkSonido;
        private CheckBox chkAutoGuardar;

        public FormCheckBox()
        {
            // Configuración del formulario
            this.Text = "Opciones de Configuración";
            this.Size = new Size(400, 250);
            this.StartPosition = FormStartPosition.CenterScreen;
            this.BackColor = Color.LightYellow;

            // CheckBox: Notificaciones
            chkNotificaciones = new CheckBox();
            chkNotificaciones.Text = "Activar notificaciones";
            chkNotificaciones.Location = new Point(50, 30);
            chkNotificaciones.CheckedChanged += CheckBox_CheckedChanged;
            chkNotificaciones.Click += CheckBox_Click;

            // CheckBox: Sonido
            chkSonido = new CheckBox();
            chkSonido.Text = "Activar sonido";
            chkSonido.Location = new Point(50, 70);
            chkSonido.CheckedChanged += CheckBox_CheckedChanged;
            chkSonido.Click += CheckBox_Click;
        }
    }
}
```

```

        // CheckBox: AutoGuardar
        chkAutoGuardar = new CheckBox();
        chkAutoGuardar.Text = "Activar guardado automático";
        chkAutoGuardar.Location = new Point(50, 110);
        chkAutoGuardar.CheckedChanged += CheckBox_CheckedChanged;
        chkAutoGuardar.Click += CheckBox_Click;

        // Agregar controles al formulario
        this.Controls.Add(chkNotificaciones);
        this.Controls.Add(chkSonido);
        this.Controls.Add(chkAutoGuardar);
    }

    // Evento CheckedChanged
    private void CheckBox_CheckedChanged(object sender, EventArgs e)
    {
        CheckBox chk = sender as CheckBox;
        if (chk.Checked)
            MessageBox.Show($"{chk.Text} activado");
        else
            MessageBox.Show($"{chk.Text} desactivado");
    }

    // Evento Click
    private void CheckBox_Click(object sender, EventArgs e)
    {
        // Registro de clic en consola
        Console.WriteLine("Se hizo clic en: " + ((CheckBox)sender).Text);
    }

    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new FormCheckBox());
    }

```

```
}  
}
```

Explicación del ejemplo

1. **Formulario:** tamaño 400x250, fondo amarillo, centrado en pantalla.
2. **CheckBox:** tres casillas para controlar diferentes funciones.
3. **Eventos:**
 - `CheckedChanged` → muestra mensaje al cambiar estado.
 - `Click` → registra el clic en consola.
4. **Métodos:** `Focus()` y `Refresh()` podrían usarse si se desea programáticamente cambiar el foco o actualizar visualmente las casillas.

Este ejemplo muestra cómo **las casillas permiten activar o desactivar funciones** en tiempo real, y cómo combinar múltiples `CheckBox` para controlar varias opciones simultáneamente.

Lección 7: Objeto RadioButton en Windows Forms

El **RadioButton** es un control que permite al usuario **seleccionar una opción de un conjunto de alternativas**. A diferencia del **CheckBox**, los **RadioButton** se usan en **grupos**, de modo que solo se puede seleccionar **una opción dentro del grupo** a la vez.

Se utiliza frecuentemente para formularios, encuestas, configuraciones o cualquier situación en la que haya que elegir **una única opción**.

7.1. Propiedades principales del RadioButton

El **RadioButton** tiene propiedades que controlan su **estado, apariencia y comportamiento**:

1. Text

- Define el **texto visible junto al RadioButton**, describiendo la opción.

```
radioButton1.Text = "Opción 1";
```

2. Checked

- Indica si el **RadioButton** está **seleccionado (true) o no (false)**.
- Solo uno de los **RadioButton** en el mismo grupo puede estar marcado.

```
radioButton1.Checked = true; // Seleccionado por defecto
```

3. Appearance

- Determina el **aspecto visual** del control:
 - Normal** → apariencia estándar de botón de radio.
 - Button** → parece un botón que se puede presionar.

```
radioButton1.Appearance = Appearance.Normal;  
// o Appearance.Button
```

7.2. Eventos más usados del RadioButton

El **RadioButton** responde principalmente a **cambios de selección y clics**:

1. CheckedChanged

- Se dispara **cada vez que cambia el estado del RadioButton**, ya sea por usuario o por código.
- Ideal para actualizar la interfaz o ejecutar acciones según la opción seleccionada.

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)  
{
```

```
        if (radioButton1.Checked)
            MessageBox.Show("Has seleccionado Opción 1");
    }
```

2. Click

- Se produce **cuando el usuario hace clic en el RadioButton**, incluso si ya estaba seleccionado.

```
private void radioButton1_Click(object sender, EventArgs e)
{
    Console.WriteLine("Se hizo clic en: " + ((RadioButton)sender).Text);
}
```

7.3. Métodos importantes del RadioButton

Aunque el RadioButton es sencillo, posee métodos útiles:

1. Focus()

- Asigna el **foco al RadioButton**, permitiendo seleccionarlo con la barra espaciadora.

```
radioButton1.Focus();
```

2. Refresh()

- Fuerza la **actualización visual** del RadioButton después de cambiar propiedades en tiempo de ejecución.

```
radioButton1.Refresh();
```

7.4. Ejemplo práctico: Selección de opciones exclusivas

Objetivo

Crear un formulario con **tres RadioButton** que permitan seleccionar **una única opción** de color, y al seleccionarla se muestre un mensaje y se cambie el color de fondo del formulario.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace RadioButtonEjemplo
{
    public class FormRadioButton : Form
    {

```

```
private RadioButton rdoRojo, rdoVerde, rdoAzul;

public FormRadioButton()
{
    // Configuración del formulario
    this.Text = "Selección de Color";
    this.Size = new Size(400, 250);
    this.StartPosition = FormStartPosition.CenterScreen;
    this.BackColor = Color.LightGray;

    // RadioButton Rojo
    rdoRojo = new RadioButton();
    rdoRojo.Text = "Rojo";
    rdoRojo.Location = new Point(50, 30);
    rdoRojo.CheckedChanged += RadioButton_CheckedChanged;
    rdoRojo.Click += RadioButton_Click;

    // RadioButton Verde
    rdoVerde = new RadioButton();
    rdoVerde.Text = "Verde";
    rdoVerde.Location = new Point(50, 70);
    rdoVerde.CheckedChanged += RadioButton_CheckedChanged;
    rdoVerde.Click += RadioButton_Click;

    // RadioButton Azul
    rdoAzul = new RadioButton();
    rdoAzul.Text = "Azul";
    rdoAzul.Location = new Point(50, 110);
    rdoAzul.CheckedChanged += RadioButton_CheckedChanged;
    rdoAzul.Click += RadioButton_Click;

    // Agregar controles al formulario
    this.Controls.Add(rdoRojo);
    this.Controls.Add(rdoVerde);
    this.Controls.Add(rdoAzul);
}
```

```

// Evento CheckedChanged
private void RadioButton_CheckedChanged(object sender, EventArgs e)
{
    RadioButton rdo = sender as RadioButton;
    if (rdo.Checked)
    {
        switch (rdo.Text)
        {
            case "Rojo":
                this.BackColor = Color.Red;
                break;
            case "Verde":
                this.BackColor = Color.Green;
                break;
            case "Azul":
                this.BackColor = Color.Blue;
                break;
        }
        MessageBox.Show("Has seleccionado: " + rdo.Text);
    }
}

// Evento Click
private void RadioButton_Click(object sender, EventArgs e)
{
    Console.WriteLine("Clic en: " + ((RadioButton)sender).Text);
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormRadioButton());
}
}

```

```
}
```

Explicación del ejemplo

1. **Formulario:** tamaño 400x250, fondo gris, centrado en pantalla.
2. **RadioButton:** tres opciones exclusivas (Rojo, Verde, Azul).
3. **Eventos:**
 - `CheckedChanged` → actualiza el color de fondo del formulario y muestra mensaje al seleccionar.
 - `Click` → registra el clic en consola.
4. **Agrupación implícita:**
 - Todos los `RadioButton` dentro del mismo formulario (o `Panel/GroupBox`) forman un **grupo automático**, permitiendo que solo uno esté seleccionado.

Este ejemplo demuestra cómo **los RadioButton permiten seleccionar opciones exclusivas** y cómo se puede interactuar con otros controles y propiedades del formulario según la selección.

Lección 8: Objeto ComboBox en Windows Forms

El **ComboBox** es un control que combina un **TextBox editable** con una **lista desplegable de elementos**, permitiendo al usuario **seleccionar un valor existente o escribir uno nuevo** (según la configuración).

Se utiliza frecuentemente para **listas de selección, formularios de entrada y menús de opciones**, y permite mantener la interfaz limpia incluso cuando hay muchas opciones disponibles.

8.1. Propiedades principales del ComboBox

El ComboBox posee propiedades que determinan su **contenido, selección y comportamiento**:

1. Text

- Contiene el **texto actual del ComboBox**.
- Puede cambiarse por código o escribirse directamente si el ComboBox es editable.

```
comboBox1.Text = "Seleccione un país";
```

2. Items

- Colección de **elementos disponibles** en la lista desplegable.
- Se puede agregar, eliminar o acceder a los elementos.

```
comboBox1.Items.Add("España");  
comboBox1.Items.Add("Francia");  
comboBox1.Items.Add("Alemania");
```

3. SelectedIndex

- Indica el **índice del elemento seleccionado** (-1 si no hay selección).
- Permite obtener o establecer el elemento seleccionado por posición.

```
comboBox1.SelectedIndex = 0; // Selecciona el primer elemento
```

4. DropDownStyle

- Define el **modo de interacción** del ComboBox:
 - **DropDown** → editable, usuario puede escribir o seleccionar.
 - **DropDownList** → no editable, solo selección de lista.
 - **Simple** → lista siempre visible, editable.

```
comboBox1.DropDownStyle = ComboBoxStyle.DropDownList;
```

5. Sorted

- Indica si los elementos se muestran **ordenados alfabéticamente**.

```
comboBox1.Sorted = true;
```

8.2. Eventos más usados del ComboBox

El ComboBox responde a cambios de selección y escritura de texto:

1. SelectedIndexChanged

- Se dispara **cada vez que cambia el elemento seleccionado** en la lista.

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show("Has seleccionado: " + comboBox1.SelectedItem.ToString());
}
```

2. TextChanged

- Se activa **cuando el texto cambia**, ya sea por selección de un elemento o escritura del usuario.

```
private void comboBox1_TextChanged(object sender, EventArgs e)
{
    lblInfo.Text = "Texto actual: " + comboBox1.Text;
}
```

3. DropDownClosed

- Se produce **cuando se cierra la lista desplegable**, útil para ejecutar acciones tras la selección o escritura final.

```
private void comboBox1_DropDownClosed(object sender, EventArgs e)
{
    Console.WriteLine("Se cerró la lista desplegable");
}
```

8.3. Métodos importantes del ComboBox

El ComboBox tiene métodos que permiten **agregar, buscar y manipular elementos**:

Método	Descripción	Ejemplo
Items.Add(object item)	Agrega un elemento a la lista	comboBox1.Items.Add("Italia");
Items.Clear()	Elimina todos los elementos de la lista	comboBox1.Items.Clear();
FindString(string s)	Devuelve el índice del primer elemento que contiene la cadena	int i = comboBox1.FindString("Fra");
ResetText()	Limpia el texto del ComboBox	comboBox1.ResetText();

8.4. Ejemplo práctico: Listado dinámico de opciones

Objetivo

Crear un formulario con un ComboBox que:

1. Permita **seleccionar o escribir** un elemento.
2. Muestre la selección en un Label.
3. Permita **agregar nuevos elementos dinámicamente** mediante un botón.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace ComboBoxEjemplo
{
    public class FormComboBox : Form
    {
        private ComboBox comboPaises;
        private Label lblSeleccion;
        private Button btnAgregar;
        private TextBox txtNuevoPais;

        public FormComboBox()
        {
            // Configuración del formulario
            this.Text = "ComboBox Dinámico";
            this.Size = new Size(400, 250);
            this.StartPosition = FormStartPosition.CenterScreen;

            // ComboBox
            comboPaises = new ComboBox();
            comboPaises.Location = new Point(50, 30);
            comboPaises.Size = new Size(200, 30);
            comboPaises.DropDownStyle = ComboBoxStyle.DropDown; // editable
            comboPaises.Sorted = true;
            comboPaises.Items.AddRange(new string[] { "España", "Francia",
"Alemania" });
        }
    }
}
```

```

        comboPaises.SelectedIndexChanged +=
ComboPaises_SelectedIndexChanged;

        comboPaises.TextChanged += ComboPaises_TextChanged;
        comboPaises.DropDownClosed += ComboPaises_DropDownClosed;

// Label
lblSeleccion = new Label();
lblSeleccion.Location = new Point(50, 70);
lblSeleccion.Size = new Size(300, 30);
lblSeleccion.Text = "Selecciona un país";

// TextBox para nuevo país
txtNuevoPais = new TextBox();
txtNuevoPais.Location = new Point(50, 110);
txtNuevoPais.Size = new Size(150, 30);

// Botón para agregar
btnAgregar = new Button();
btnAgregar.Text = "Agregar país";
btnAgregar.Location = new Point(210, 110);
btnAgregar.Click += BtnAgregar_Click;

// Agregar controles al formulario
this.Controls.Add(comboPaises);
this.Controls.Add(lblSeleccion);
this.Controls.Add(txtNuevoPais);
this.Controls.Add(btnAgregar);
}

e) private void ComboPaises_SelectedIndexChanged(object sender, EventArgs
{
    if (comboPaises.SelectedItem != null)
        lblSeleccion.Text = "Seleccionaste: " +
comboPaises.SelectedItem.ToString();
}

```

```

private void ComboPaises_TextChanged(object sender, EventArgs e)
{
    lblSeleccion.Text = "Texto actual: " + comboPaises.Text;
}

private void ComboPaises_DropDownClosed(object sender, EventArgs e)
{
    Console.WriteLine("Se cerró la lista desplegable");
}

private void BtnAgregar_Click(object sender, EventArgs e)
{
    string nuevoPais = txtNuevoPais.Text.Trim();
    if (!string.IsNullOrEmpty(nuevoPais))
    {
        comboPaises.Items.Add(nuevoPais);
        comboPaises.ResetText();
        txtNuevoPais.Clear();
    }
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormComboBox());
}
}

```

Explicación del ejemplo

1. **Formulario:** tamaño 400x250, centrado en pantalla.
2. **ComboBox:**
 - Editable (DropDown), ordenado alfabéticamente (Sorted = true).
 - Contiene elementos iniciales: España, Francia, Alemania.
 - Eventos: SelectedIndexChanged, TextChanged, DropDownClosed.

3. **Label:** muestra la selección actual en tiempo real.
4. **TextBox y Botón:** permiten **agregar elementos dinámicamente** al ComboBox.
5. **Métodos usados:** `Items.Add()`, `ResetText()`, `Items.Clear()` en caso de reiniciar la lista.

Este ejemplo muestra cómo **un ComboBox puede ser dinámico**, editable y totalmente interactivo, integrando selección de elementos y agregando nuevos valores en tiempo de ejecución.

Lección 9: Objeto ListBox en Windows Forms

El **ListBox** es un control que permite **mostrar una lista de elementos** de forma vertical (o en columnas), y seleccionar uno o varios elementos dependiendo de la configuración.

Es ampliamente utilizado para **listas de tareas, selección múltiple, menús y almacenamiento temporal de datos** dentro de una interfaz de Windows Forms.

9.1. Propiedades principales del ListBox

El ListBox posee propiedades que determinan su **contenido, selección y presentación**:

1. Items

- Colección de elementos contenidos en el ListBox.
- Permite agregar, eliminar o acceder a los elementos.

```
listBox1.Items.Add("Tarea 1");
```

```
listBox1.Items.Add("Tarea 2");
```

2. SelectedIndex

- Devuelve o establece el **índice del elemento actualmente seleccionado** (-1 si no hay selección).

```
int indice = listBox1.SelectedIndex;
```

```
listBox1.SelectedIndex = 0; // Selecciona el primer elemento
```

3. MultiColumn

- Si se establece en `true`, los elementos se muestran en **varias columnas** si la lista es larga.

```
listBox1.MultiColumn = true;
```

4. SelectionMode

- Define **cómo se puede seleccionar elementos** en la lista:
 - `One` → solo un elemento puede seleccionarse.
 - `MultiSimple` → múltiples elementos se pueden seleccionar con clic directo.
 - `MultiExtended` → selección múltiple usando Shift o Ctrl.

```
listBox1.SelectionMode = SelectionMode.MultiExtended;
```

9.2. Eventos más usados del ListBox

El ListBox responde a cambios de selección y acciones del usuario:

1. SelectedIndexChanged

- Se dispara **cuando cambia el elemento seleccionado**, ya sea individual o múltiple.

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (listBox1.SelectedItem != null)
        lblInfo.Text = "Seleccionaste: " + listBox1.SelectedItem.ToString();
}
```

2. DoubleClick

- Se produce **cuando el usuario hace doble clic sobre un elemento**, útil para ejecutar acciones rápidas.

```
private void listBox1_DoubleClick(object sender, EventArgs e)
{
    if (listBox1.SelectedItem != null)
        MessageBox.Show("Elemento doble clic: " +
listBox1.SelectedItem.ToString());
}
```

9.3. Métodos importantes del ListBox

El ListBox tiene métodos que permiten **manipular la lista de manera dinámica**:

Método	Descripción	Ejemplo
Items.Add(object item)	Agrega un elemento a la lista	listBox1.Items.Add("Tarea nueva");
Items.Remove(object item)	Elimina un elemento específico	listBox1.Items.Remove("Tarea 1");
ClearSelected()	Desmarca todos los elementos seleccionados	listBox1.ClearSelected();
FindString(string s)	Devuelve el índice del primer elemento que contiene la cadena	int i = listBox1.FindString("Tarea");

9.4. Ejemplo práctico: Lista de tareas con eliminación dinámica

Objetivo

Crear un formulario con un ListBox que permita:

1. **Agregar tareas** desde un TextBox.
2. **Eliminar tareas seleccionadas** mediante un botón.
3. Mostrar información de la tarea seleccionada.

Código completo

```
using System;
```

```
using System.Drawing;
using System.Windows.Forms;

namespace ListBoxEjemplo
{
    public class FormListBox : Form
    {
        private ListBox listTareas;
        private TextBox txtNuevaTarea;
        private Button btnAgregar, btnEliminar;
        private Label lblInfo;

        public FormListBox()
        {
            // Configuración del formulario
            this.Text = "Lista de Tareas";
            this.Size = new Size(400, 300);
            this.StartPosition = FormStartPosition.CenterScreen;

            // ListBox
            listTareas = new ListBox();
            listTareas.Location = new Point(50, 30);
            listTareas.Size = new Size(200, 150);
            listTareas.SelectionMode = SelectionMode.MultiExtended;
            listTareas.SelectedIndexChanged += ListTareas_SelectedIndexChanged;
            listTareas.DoubleClick += ListTareas_DoubleClick;

            // TextBox
            txtNuevaTarea = new TextBox();
            txtNuevaTarea.Location = new Point(50, 190);
            txtNuevaTarea.Size = new Size(150, 30);

            // Botón Agregar
            btnAgregar = new Button();
            btnAgregar.Text = "Agregar";
            btnAgregar.Location = new Point(210, 190);
```

```

        btnAgregar.Click += BtnAgregar_Click;

        // Botón Eliminar
        btnEliminar = new Button();
        btnEliminar.Text = "Eliminar seleccionados";
        btnEliminar.Location = new Point(50, 230);
        btnEliminar.Click += BtnEliminar_Click;

        // Label informativo
        lblInfo = new Label();
        lblInfo.Location = new Point(50, 270);
        lblInfo.Size = new Size(300, 20);
        lblInfo.Text = "Selecciona una tarea";

        // Agregar controles al formulario
        this.Controls.Add(listTareas);
        this.Controls.Add(txtNuevaTarea);
        this.Controls.Add(btnAgregar);
        this.Controls.Add(btnEliminar);
        this.Controls.Add(lblInfo);
    }

    // Evento SelectedIndexChanged
    private void ListTareas_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (listTareas.SelectedItem != null)
            lblInfo.Text = "Seleccionaste: " +
listTareas.SelectedItem.ToString();
    }

    // Evento DoubleClick
    private void ListTareas_DoubleClick(object sender, EventArgs e)
    {
        if (listTareas.SelectedItem != null)
            MessageBox.Show("Tarea doble clic: " +
listTareas.SelectedItem.ToString());
    }

```

```

// Agregar tarea
private void BtnAgregar_Click(object sender, EventArgs e)
{
    string tarea = txtNuevaTarea.Text.Trim();
    if (!string.IsNullOrEmpty(tarea))
    {
        listTareas.Items.Add(tarea);
        txtNuevaTarea.Clear();
    }
}

// Eliminar tareas seleccionadas
private void BtnEliminar_Click(object sender, EventArgs e)
{
    while (listTareas.SelectedItems.Count > 0)
    {
        listTareas.Items.Remove(listTareas.SelectedItems[0]);
    }
}

[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new FormListBox());
}
}

```

Explicación del ejemplo

1. **Formulario:** tamaño 400x300, centrado en pantalla.
2. **ListBox:** permite selección múltiple (`MultiExtended`) y captura eventos `SelectedIndexChanged` y `DoubleClick`.
3. **Agregar tareas:** se ingresan mediante el `TextBox` y se añaden con el botón `Agregar`.
4. **Eliminar tareas:** se eliminan todas las seleccionadas con el botón `Eliminar`.

5. **Label:** muestra información del elemento seleccionado en tiempo real.

6. **Métodos usados:** `Items.Add()`, `Items.Remove()`, `ClearSelected()`, `FindString()` se pueden aplicar para gestión dinámica de la lista.

Este ejemplo muestra cómo **un ListBox permite administrar dinámicamente elementos** con selección múltiple, actualización visual y eventos interactivos, siendo ideal para listas de tareas o inventarios en Windows Forms.

Lección 10: Objeto PictureBox en Windows Forms

El **PictureBox** es un control utilizado para **mostrar imágenes** dentro de un formulario de Windows Forms. Es muy versátil, permitiendo trabajar con imágenes locales, archivos externos o recursos embebidos. Además, permite interactuar con eventos del mouse para crear galerías, efectos visuales o interfaces interactivas.

10.1. Propiedades principales del PictureBox

El PictureBox posee propiedades que determinan su **contenido, apariencia y comportamiento**:

1. Image

- Contiene la **imagen que se mostrará** en el control.
- Puede asignarse mediante archivos externos o recursos embebidos.

```
pictureBox1.Image = Image.FromFile("imagen.jpg");
```

2. SizeMode

- Define cómo se muestra la imagen dentro del control:
 - `Normal` → posición original, recortando si es más grande que el control.
 - `StretchImage` → estira la imagen para llenar el PictureBox.
 - `AutoSize` → ajusta el tamaño del control al tamaño de la imagen.
 - `CenterImage` → centra la imagen sin cambiar su tamaño.
 - `Zoom` → ajusta la imagen al control manteniendo proporción.

```
pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
```

3. BorderStyle

- Permite agregar un **borde al PictureBox**:
 - `None` → sin borde.
 - `FixedSingle` → borde fino.
 - `Fixed3D` → borde con efecto 3D.

```
pictureBox1.BorderStyle = BorderStyle.FixedSingle;
```

4. BackColor

- Color de fondo del PictureBox, útil cuando la imagen no ocupa todo el control.

```
pictureBox1.BackColor = Color.LightGray;
```

10.2. Eventos más usados del PictureBox

El PictureBox puede reaccionar a eventos del **mouse y del usuario**:

1. Click

- Se dispara **cuando el usuario hace clic sobre la imagen**.

```
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Imagen seleccionada");
}
```

2. MouseEnter

- Ocurre **cuando el cursor entra en el área del PictureBox**, útil para efectos de resaltado.

```
private void pictureBox1_MouseEnter(object sender, EventArgs e)
{
    pictureBox1.BorderStyle = BorderStyle.Fixed3D;
}
```

3. MouseLeave

- Se produce **cuando el cursor sale del control**, para revertir cambios visuales.

```
private void pictureBox1_MouseLeave(object sender, EventArgs e)
{
    pictureBox1.BorderStyle = BorderStyle.FixedSingle;
}
```

4. MouseDown

- Se activa **cuando se presiona un botón del mouse** sobre la imagen, útil para interacciones más complejas.

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    Console.WriteLine("Botón presionado: " + e.Button);
}
```

10.3. Métodos importantes del PictureBox

El PictureBox dispone de métodos que permiten **cargar y manipular imágenes dinámicamente**:

1. Load()

- Carga una imagen desde un **archivo externo** de manera directa.

```
pictureBox1.Load("imagen.jpg");
```

2. Refresh()

- Fuerza la **actualización visual** del PictureBox después de cambios de imagen o propiedades.

```
pictureBox1.Refresh();
```

3. Dispose()

- Libera los **recursos de la imagen** y del control, útil al cerrar formularios o cambiar imágenes dinámicamente.

```
pictureBox1.Dispose();
```

10.4. Ejemplo práctico: Galería de imágenes con clic interactivo

Objetivo

Crear un formulario con varios PictureBox que actúen como **miniaturas de una galería**, mostrando la imagen seleccionada en un PictureBox principal al hacer clic.

Código completo

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace PictureBoxEjemplo
{
    public class FormGaleria : Form
    {
        private PictureBox picPrincipal;
        private PictureBox picThumb1, picThumb2, picThumb3;

        public FormGaleria()
        {
            // Configuración del formulario
            this.Text = "Galería de Imágenes";
            this.Size = new Size(600, 400);
            this.StartPosition = FormStartPosition.CenterScreen;

            // PictureBox principal
            picPrincipal = new PictureBox();
            picPrincipal.Location = new Point(50, 20);
```

```

        picPrincipal.Size = new Size(500, 250);
        picPrincipal.BorderStyle = BorderStyle.FixedSingle;
        picPrincipal.SizeMode = PictureBoxSizeMode.Zoom;

        // Miniaturas
        picThumb1 = CrearThumbnail("img1.jpg", new Point(50, 280));
        picThumb2 = CrearThumbnail("img2.jpg", new Point(200, 280));
        picThumb3 = CrearThumbnail("img3.jpg", new Point(350, 280));

        // Agregar controles al formulario
        this.Controls.Add(picPrincipal);
        this.Controls.Add(picThumb1);
        this.Controls.Add(picThumb2);
        this.Controls.Add(picThumb3);
    }

    // Método auxiliar para crear miniaturas
    private PictureBox CrearThumbnail(string ruta, Point ubicacion)
    {
        PictureBox pic = new PictureBox();
        pic.Size = new Size(120, 80);
        pic.Location = ubicacion;
        pic.BorderStyle = BorderStyle.FixedSingle;
        pic.SizeMode = PictureBoxSizeMode.Zoom;
        pic.Image = Image.FromFile(ruta);
        pic.Click += (s, e) => picPrincipal.Image = pic.Image;
        pic.MouseEnter += (s, e) => pic.BorderStyle = BorderStyle.Fixed3D;
        pic.MouseLeave += (s, e) => pic.BorderStyle =
BorderStyle.FixedSingle;
        return pic;
    }

    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new FormGaleria());
    }

```

```
    }  
  }  
}
```

Explicación del ejemplo

1. **Formulario:** tamaño 600x400, centrado en pantalla.
2. **PictureBox principal:** muestra la imagen seleccionada de la galería.
3. **Miniaturas:** tres PictureBox pequeños que actúan como previsualización.
4. **Eventos:**
 - `Click` → actualiza la imagen principal con la miniatura seleccionada.
 - `MouseEnter` / `MouseLeave` → efecto visual de borde al pasar el cursor.
5. **Método auxiliar:** `CrearThumbnail()` evita repetir código y aplica estilos uniformes a cada miniatura.

Este ejemplo demuestra cómo **PictureBox** permite crear galerías interactivas, cargar imágenes dinámicamente y manejar eventos del mouse para mejorar la experiencia visual en Windows Forms.

Lección 11: El objeto Timer en C#

El **objeto Timer** es una herramienta fundamental en C# para ejecutar código de manera periódica sin bloquear la interfaz de usuario. Permite que ciertas acciones se repitan después de un intervalo de tiempo determinado. Se utiliza mucho en **relojes digitales, animaciones automáticas, actualizaciones periódicas de datos, temporizadores de juegos**, etc.

11.1 Propiedades del Timer

1. Interval

- Tipo: `int` (milisegundos)
- Descripción: Define el tiempo que debe transcurrir entre la ejecución de cada evento `Tick`.
- Valores:
 - Expresado en **milisegundos**.
 - Ejemplo: `1000` → 1 segundo.
- Uso:

```
Timer miTimer = new Timer();  
miTimer.Interval = 1000; // Cada 1000 ms (1 segundo)
```

- Nota: Si `Interval` se establece en un valor demasiado bajo (menos de 1 ms), el sistema no garantiza precisión absoluta, ya que depende del scheduler de Windows.

2. Enabled

- Tipo: `bool`
- Descripción: Determina si el Timer está **activo** o no.
- Valores:
 - `true`: El Timer está funcionando y se ejecuta cada intervalo definido.
 - `false`: El Timer está detenido.
- Uso:

```
miTimer.Enabled = true; // Activa el Timer  
miTimer.Enabled = false; // Detiene el Timer
```

- Nota: Cambiar `Enabled` a `true` es equivalente a llamar al método `Start()`.
-

11.2 Eventos del Timer

1. Tick

- Descripción: Este evento se dispara **cada vez que transcurre el intervalo definido en Interval**.
- Se utiliza para ejecutar cualquier acción repetitiva.
- Ejemplo de suscripción:

```
miTimer.Tick += MiTimer_Tick;
```

```
private void MiTimer_Tick(object sender, EventArgs e)
{
    // Código que se ejecuta cada segundo
    Console.WriteLine("¡Ha pasado un segundo!");
}
```

- Nota: El evento `Tick` se ejecuta en el **hilo principal de la interfaz**, por lo que es seguro actualizar controles de Windows Forms directamente desde él.

11.3 Métodos del Timer

1. Start()

- Inicia el Timer.
- Equivalente a `Enabled = true`.

```
miTimer.Start();
```

2. Stop()

- Detiene el Timer.
- Equivalente a `Enabled = false`.

```
miTimer.Stop();
```

3. Dispose()

- Libera los recursos utilizados por el Timer.
- Es una buena práctica usarlo cuando el Timer ya no se necesita.

```
miTimer.Dispose();
```

11.4 Ejemplos prácticos

Ejemplo 1: Reloj digital

Un reloj que se actualiza cada segundo en un `Label` de Windows Forms.

```
public partial class Form1 : Form
{
    Timer relojTimer;

    public Form1()
    {
        InitializeComponent();

        relojTimer = new Timer();
        relojTimer.Interval = 1000; // 1 segundo
        relojTimer.Tick += RelojTimer_Tick;
        relojTimer.Start();
    }

    private void RelojTimer_Tick(object sender, EventArgs e)
    {
        lblReloj.Text = DateTime.Now.ToString("HH:mm:ss");
    }
}
```

- Resultado: El `Label lblReloj` muestra la hora actual y se actualiza cada segundo automáticamente.

Ejemplo 2: Animación automática

Mover un botón de izquierda a derecha de manera automática en un formulario.

```
public partial class Form1 : Form
{
    Timer animTimer;
    int velocidad = 5;

    public Form1()
    {
        InitializeComponent();
    }
}
```

```

        animTimer = new Timer();
        animTimer.Interval = 50; // 50 ms
        animTimer.Tick += AnimTimer_Tick;
        animTimer.Start();
    }

    private void AnimTimer_Tick(object sender, EventArgs e)
    {
        btnMover.Left += velocidad;

        // Rebotar cuando llega al borde
        if (btnMover.Right >= this.ClientSize.Width || btnMover.Left <= 0)
        {
            velocidad = -velocidad;
        }
    }
}

```

- Resultado: El botón se mueve de izquierda a derecha de forma continua, cambiando de dirección al tocar los bordes.

Resumen

Concepto	Descripción
Interval	Tiempo entre cada ejecución del <code>Tick</code> (ms)
Enabled	Activa o desactiva el <code>Timer</code>
Tick	Evento que se dispara cada <code>Interval</code>
Start()	Inicia el <code>Timer</code>
Stop()	Detiene el <code>Timer</code>
Dispose()	Libera los recursos del <code>Timer</code>

El `Timer` es esencial para tareas periódicas y animaciones ligeras. Su ventaja principal es que **no bloquea la interfaz**, permitiendo que la aplicación siga siendo interactiva mientras se ejecuta código repetitivo.

Tercera Parte: Controles de Contenedor y Organización

Los **controles de contenedor** son aquellos controles que permiten agrupar y organizar otros controles dentro de una interfaz gráfica. Su objetivo principal es **mejorar la estructura visual, la claridad y la lógica del formulario**. Entre los más usados se encuentran **GroupBox** y **Panel**.

Estos controles son fundamentales cuando se desea que la interfaz tenga **secciones diferenciadas**, facilitando la comprensión tanto para el usuario como para el desarrollador.

Leccion 12: Objeto GroupBox y Panel

12.1 Propiedades principales

a) Text

- **Descripción:** Define el texto que se muestra como título del contenedor.
- **Uso práctico:** Permite identificar de manera clara la sección de controles que agrupa.
- **Ejemplo:**

```
groupBox1.Text = "Datos Personales";
```

b) BackColor

- **Descripción:** Cambia el color de fondo del contenedor.
- **Uso práctico:** Diferencia visualmente grupos de controles dentro del formulario.
- **Ejemplo:**

```
panel1.BackColor = Color.LightGray;
```

c) Enabled

- **Descripción:** Permite habilitar o deshabilitar todos los controles contenidos en el contenedor.
- **Uso práctico:** Bloquear temporalmente una sección del formulario.
- **Ejemplo:**

```
groupBox1.Enabled = false; // Todos los controles quedan deshabilitados
```

d) Dock

- **Descripción:** Determina cómo el contenedor se ajusta automáticamente al formulario o contenedor padre.
- **Opciones comunes:** Top, Bottom, Left, Right, Fill.
- **Ejemplo:**

```
panel1.Dock = DockStyle.Fill; // Ocupa todo el espacio disponible
```

e) Anchor

- **Descripción:** Mantiene la posición relativa del contenedor al tamaño del formulario.
- **Uso práctico:** Permite que los controles mantengan su posición al redimensionar el formulario.
- **Ejemplo:**

```
groupBox1.Anchor = AnchorStyles.Top | AnchorStyles.Left;
```

12.2 Métodos importantes

a) Refresh()

- **Descripción:** Fuerza la actualización visual del contenedor y sus controles.
- **Uso práctico:** Cuando se realizan cambios dinámicos en tiempo de ejecución y se desea reflejarlos inmediatamente.
- **Ejemplo:**

```
panel1.Refresh();
```

b) Focus()

- **Descripción:** Establece el foco en el contenedor o en un control específico dentro del mismo.
- **Uso práctico:** Dirigir la atención del usuario hacia un grupo de controles.
- **Ejemplo:**

```
groupBox1.Focus();
```

c) Controls.Add()

- **Descripción:** Permite agregar controles hijos de manera dinámica al contenedor.
- **Uso práctico:** Ideal para interfaces donde los controles se crean en tiempo de ejecución.
- **Ejemplo:**

```
Button btn = new Button();  
btn.Text = "Aceptar";  
panel1.Controls.Add(btn);
```

12.3 Ejemplo práctico: Agrupación lógica de controles

Supongamos que queremos agrupar los controles relacionados con los **datos de contacto de un usuario**:

```
// Crear el GroupBox  
GroupBox groupBoxContacto = new GroupBox();  
groupBoxContacto.Text = "Datos de Contacto";  
groupBoxContacto.Size = new Size(300, 150);
```

```
groupBoxContacto.Location = new Point(10, 10);
```

```
// Crear controles dentro del GroupBox
```

```
Label lblEmail = new Label();
```

```
lblEmail.Text = "Email:";
```

```
lblEmail.Location = new Point(10, 30);
```

```
TextBox txtEmail = new TextBox();
```

```
txtEmail.Location = new Point(80, 30);
```

```
txtEmail.Width = 200;
```

```
Label lblTelefono = new Label();
```

```
lblTelefono.Text = "Teléfono:";
```

```
lblTelefono.Location = new Point(10, 70);
```

```
TextBox txtTelefono = new TextBox();
```

```
txtTelefono.Location = new Point(80, 70);
```

```
txtTelefono.Width = 200;
```

```
// Agregar controles al GroupBox
```

```
groupBoxContacto.Controls.Add(lblEmail);
```

```
groupBoxContacto.Controls.Add(txtEmail);
```

```
groupBoxContacto.Controls.Add(lblTelefono);
```

```
groupBoxContacto.Controls.Add(txtTelefono);
```

```
// Agregar GroupBox al formulario
```

```
this.Controls.Add(groupBoxContacto);
```

```
// Crear un Panel adicional
```

```
Panel panelBotones = new Panel();
```

```
panelBotones.Size = new Size(300, 50);
```

```
panelBotones.Location = new Point(10, 170);
```

```
panelBotones.BackColor = Color.LightBlue;
```

```
// Crear botones dentro del Panel
```

```
Button btnGuardar = new Button();
```

```
btnGuardar.Text = "Guardar";  
btnGuardar.Location = new Point(10, 10);  
  
Button btnCancelar = new Button();  
btnCancelar.Text = "Cancelar";  
btnCancelar.Location = new Point(100, 10);  
  
// Agregar botones al Panel  
panelBotones.Controls.Add(btnGuardar);  
panelBotones.Controls.Add(btnCancelar);  
  
// Agregar Panel al formulario  
this.Controls.Add(panelBotones);
```

Explicación:

- Se crea un **GroupBox** para los datos de contacto y se agregan controles relacionados dentro de él.
- Se utiliza un **Panel** para contener botones de acción.
- Esta organización permite **claridad visual y lógica**, diferenciando secciones del formulario y facilitando futuras modificaciones o ampliaciones.

Beneficios de usar GroupBox y Panel

1. **Organización visual:** Facilita la lectura de la interfaz.
 2. **Agrupación lógica:** Los controles relacionados se manejan juntos.
 3. **Flexibilidad:** Se pueden habilitar, deshabilitar o mover secciones completas fácilmente.
 4. **Reutilización:** Contenedores pueden usarse como bloques funcionales en distintos formularios.
-

Leccion 13: Objeto TabControl

El **TabControl** es un **control contenedor** que permite organizar la interfaz en **pestañas**, cada una de las cuales puede contener otros controles. Su principal ventaja es **mostrar gran cantidad de información sin saturar el formulario**, dividiéndola en secciones claramente diferenciadas.

Es ampliamente usado en formularios complejos, donde se requiere **navegar entre distintas categorías o secciones de datos**.

13.1 Propiedades principales

a) TabPages

- **Descripción:** Colección que contiene todas las pestañas del control.
- **Uso práctico:** Permite agregar, eliminar o recorrer pestañas programáticamente.
- **Ejemplo:**

```
tabControl1.TabPages.Add(tabPage1);  
tabControl1.TabPages.Add(tabPage2);
```

b) SelectedIndex

- **Descripción:** Índice de la pestaña actualmente activa.
- **Uso práctico:** Se puede cambiar la pestaña activa mediante programación o consultar cuál está activa.
- **Ejemplo:**

```
// Selecciona la primera pestaña  
tabControl1.SelectedIndex = 0;  
  
// Consultar la pestaña activa  
int indiceActual = tabControl1.SelectedIndex;
```

c) Alignment

- **Descripción:** Define la posición de las pestañas dentro del control.
- **Opciones comunes:** Top, Bottom, Left, Right.
- **Uso práctico:** Permite personalizar la ubicación de las pestañas según el diseño de la interfaz.
- **Ejemplo:**

```
tabControl1.Alignment = TabAlignment.Top; // Pestañas en la parte superior
```

13.2 Eventos importantes

SelectedIndexChanged

- **Descripción:** Se dispara cuando el usuario cambia de pestaña.
- **Uso práctico:** Permite ejecutar código específico según la pestaña seleccionada.
- **Ejemplo:**

```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show("Pestaña seleccionada: " + tabControl1.SelectedIndex);
}
```

- Este evento es útil para **cargar datos dinámicos** o **actualizar controles** cuando se cambia de pestaña.
-

13.3 Métodos importantes

a) TabPages.Add()

- **Descripción:** Agrega una nueva pestaña al TabControl.
- **Ejemplo:**

```
TabPage nuevaPestaña = new TabPage("Nueva Pestaña");
tabControl1.TabPages.Add(nuevaPestaña);
```

b) TabPages.Remove()

- **Descripción:** Elimina una pestaña existente del TabControl.
- **Ejemplo:**

```
tabControl1.TabPages.Remove(tabPage2);
```

13.4 Ejemplo práctico: Aplicación con pestañas

Vamos a crear un formulario con **tres pestañas**, cada una con controles diferentes:

```
// Crear TabControl
TabControl tabControl1 = new TabControl();
tabControl1.Size = new Size(400, 200);
tabControl1.Location = new Point(10, 10);
tabControl1.Alignment = TabAlignment.Top;

// Crear TabPages
TabPage tabPage1 = new TabPage("Personal");
```

```
TabPage tabPage2 = new TabPage("Contacto");
TabPage tabPage3 = new TabPage("Notas");

// Agregar controles a la primera pestaña (Personal)
Label lblNombre = new Label();
lblNombre.Text = "Nombre:";
lblNombre.Location = new Point(10, 20);
TextBox txtNombre = new TextBox();
txtNombre.Location = new Point(80, 20);
tabPage1.Controls.Add(lblNombre);
tabPage1.Controls.Add(txtNombre);

// Agregar controles a la segunda pestaña (Contacto)
Label lblEmail = new Label();
lblEmail.Text = "Email:";
lblEmail.Location = new Point(10, 20);
TextBox txtEmail = new TextBox();
txtEmail.Location = new Point(80, 20);
tabPage2.Controls.Add(lblEmail);
tabPage2.Controls.Add(txtEmail);

// Agregar controles a la tercera pestaña (Notas)
Label lblNotas = new Label();
lblNotas.Text = "Notas:";
lblNotas.Location = new Point(10, 20);
TextBox txtNotas = new TextBox();
txtNotas.Location = new Point(10, 50);
txtNotas.Size = new Size(360, 100);
txtNotas.Multiline = true;
tabPage3.Controls.Add(lblNotas);
tabPage3.Controls.Add(txtNotas);

// Agregar pestañas al TabControl
tabControl1.TabPages.Add(tabPage1);
tabControl1.TabPages.Add(tabPage2);
tabControl1.TabPages.Add(tabPage3);
```

```
// Agregar TabControl al formulario
this.Controls.Add(tabControl1);

// Evento SelectedIndexChanged
tabControl1.SelectedIndexChanged += (s, e) =>
{
    MessageBox.Show("Pestaña actual: " + tabControl1.SelectedTab.Text);
};
```

Explicación paso a paso:

1. Se crea un `TabControl` y se define tamaño, ubicación y alineación de pestañas.
2. Se crean tres `TabPage` con nombres descriptivos.
3. Se agregan controles a cada pestaña según su temática.
4. Se añaden las pestañas al `TabControl` y finalmente al formulario.
5. Se maneja el evento `SelectedIndexChanged` para mostrar un mensaje al cambiar de pestaña.

Beneficios del TabControl:

- Divide la información en secciones claras y manejables.
 - Facilita la navegación y mejora la experiencia del usuario.
 - Permite carga y actualización dinámica de controles según la pestaña activa.
-

Lección 14: Objeto SplitContainer y FlowLayoutPanel

Los **SplitContainer** y **FlowLayoutPanel** son controles de contenedor avanzados que permiten crear **interfaces flexibles y adaptables**, facilitando la organización de los controles y su comportamiento al redimensionar el formulario.

Estos controles son muy útiles para **diseños dinámicos**, donde se requiere que la interfaz se ajuste automáticamente al tamaño de la ventana o cambie su distribución según el espacio disponible.

14.1 Propiedades principales

a) SplitContainer

El **SplitContainer** divide su área en dos paneles (Panel1 y Panel2), separados por un divisor ajustable por el usuario o mediante programación.

- **Orientation**

- Define la orientación del divisor.
- Opciones: `Orientation.Horizontal` (divisor horizontal, Panel1 arriba y Panel2 abajo), `Orientation.Vertical` (divisor vertical, Panel1 izquierda y Panel2 derecha).

- **Ejemplo:**

```
splitContainer1.Orientation = Orientation.Vertical;
```

- **SplitterDistance**

- Determina la distancia inicial del divisor desde el borde del control.
- **Ejemplo:**

```
splitContainer1.SplitterDistance = 200; // Panel1 tendrá 200 píxeles
```

- **Panel1 y Panel2**

- Permiten acceder a cada panel para agregar controles.

- **Ejemplo:**

```
splitContainer1.Panel1.Controls.Add(button1);  
splitContainer1.Panel2.Controls.Add(textBox1);
```

b) FlowLayoutPanel

El **FlowLayoutPanel** organiza automáticamente los controles en **una fila o columna**, ajustándose al tamaño del contenedor.

- **FlowDirection**

- Define la dirección en que se colocan los controles: `LeftToRight`, `TopDown`, `RightToLeft`, `BottomUp`.

- **Ejemplo:**

```
flowLayoutPanel1.FlowDirection = FlowDirection.LeftToRight;
```

- **WrapContents**

- Determina si los controles deben ajustarse automáticamente a la siguiente fila o columna cuando se alcanza el borde del panel.

- **Ejemplo:**

```
flowLayoutPanel1.WrapContents = true;
```

- **AutoScroll**

- Permite que el panel muestre barras de desplazamiento si los controles exceden su tamaño.

- **Ejemplo:**

```
flowLayoutPanel1.AutoScroll = true;
```

14.2 Eventos importantes

SplitterMoved (SplitContainer)

- **Descripción:** Se dispara cuando el usuario mueve el divisor.
- **Uso práctico:** Permite ajustar dinámicamente el tamaño de los controles o realizar acciones según la posición del divisor.
- **Ejemplo:**

```
private void splitContainer1_SplitterMoved(object sender, SplitterEventArgs e)
{
    label1.Text = "Splitter en posición: " + splitContainer1.SplitterDistance;
}
```

FlowLayoutPanel

- Normalmente no requiere eventos especiales para su disposición automática, pero se pueden usar eventos de los controles contenidos para interacción dinámica.

14.3 Ejemplo práctico: Diseño adaptable a redimensionamiento

Supongamos que queremos un formulario con un **panel lateral de botones** y un **área principal de contenido**, que se adapte al redimensionamiento de la ventana:

```
// Crear SplitContainer
```

```

SplitContainer splitContainer1 = new SplitContainer();
splitContainer1.Dock = DockStyle.Fill;
splitContainer1.Orientation = Orientation.Vertical;
splitContainer1.SplitterDistance = 150;

// Crear FlowLayoutPanel para botones en Panel1
FlowLayoutPanel flowLayoutPanel1 = new FlowLayoutPanel();
flowLayoutPanel1.Dock = DockStyle.Fill;
flowLayoutPanel1.FlowDirection = FlowDirection.TopDown;
flowLayoutPanel1.WrapContents = true;
flowLayoutPanel1.AutoScroll = true;

// Crear botones dinámicamente
for (int i = 1; i <= 5; i++)
{
    Button btn = new Button();
    btn.Text = "Botón " + i;
    btn.Width = 120;
    flowLayoutPanel1.Controls.Add(btn);
}

// Agregar FlowLayoutPanel al Panel1 del SplitContainer
splitContainer1.Panel1.Controls.Add(flowLayoutPanel1);

// Agregar TextBox de ejemplo al Panel2
TextBox txtContenido = new TextBox();
txtContenido.Multiline = true;
txtContenido.Dock = DockStyle.Fill;
splitContainer1.Panel2.Controls.Add(txtContenido);

// Evento SplitterMoved
splitContainer1.SplitterMoved += (s, e) =>
{
    txtContenido.Text = "Splitter moved. Nueva distancia: " +
splitContainer1.SplitterDistance;
};

```

```
// Agregar SplitContainer al formulario  
this.Controls.Add(splitContainer1);
```

Explicación paso a paso:

1. Se crea un `SplitContainer` vertical que divide el formulario en **panel lateral** y **área principal**.
2. En el `Panel1`, se coloca un `FlowLayoutPanel` con botones que se ajustan automáticamente y permiten desplazamiento si se agregan más controles.
3. En el `Panel2`, un `TextBox` ocupa todo el espacio disponible y se adapta al tamaño del panel.
4. Se maneja el evento `SplitterMoved` para mostrar la posición actual del divisor.
5. Al redimensionar la ventana, el contenido se ajusta automáticamente, manteniendo una **interfaz flexible y organizada**.

Beneficios de usar `SplitContainer` y `FlowLayoutPanel`

- Permiten **interfaces adaptables y dinámicas**.
 - Facilitan la organización de controles sin necesidad de calcular posiciones manualmente.
 - Mejoran la **experiencia de usuario** al mantener una disposición clara al cambiar el tamaño del formulario.
 - Combinables con otros contenedores como `GroupBox`, `Panel` y `TabControl` para crear interfaces complejas y profesionales.
-

Cuarta Parte: Controles de Datos y Avanzados

El **DataGridView** es uno de los controles más importantes en Windows Forms cuando se trabaja con **datos tabulares**. Permite mostrar información en **filas y columnas**, y ofrece capacidades avanzadas de **edición, selección y vinculación a fuentes de datos**, lo que lo hace ideal para aplicaciones que manejan inventarios, registros, reportes y formularios complejos.

Lección 15: Objeto DataGridView

15.1 Propiedades principales

a) DataSource

- **Descripción:** Permite enlazar el DataGridView a una **fuentes de datos** como `DataTable`, `BindingList` o colecciones de objetos.
- **Uso práctico:** Facilita mostrar y actualizar datos de manera automática.
- **Ejemplo:**

```
dataGridView1.DataSource = miDataTable;
```

b) Columns

- **Descripción:** Colección que contiene las columnas del DataGridView.
- **Uso práctico:** Permite agregar, eliminar o modificar columnas de manera manual.
- **Ejemplo:**

```
dataGridView1.Columns.Add("Producto", "Producto");
```

```
dataGridView1.Columns.Add("Precio", "Precio");
```

c) Rows

- **Descripción:** Colección de filas del DataGridView.
- **Uso práctico:** Permite agregar, eliminar o recorrer filas de manera programática.
- **Ejemplo:**

```
dataGridView1.Rows.Add("Manzanas", 1.25);
```

d) ReadOnly

- **Descripción:** Indica si las celdas son **solo lectura** o editables.
- **Uso práctico:** Bloquea ciertas columnas o el control completo para evitar cambios accidentales.
- **Ejemplo:**

```
dataGridView1.ReadOnly = false; // Permite edición
```

e) SelectionMode

- **Descripción:** Determina cómo se seleccionan las celdas o filas.
- **Opciones comunes:** FullRowSelect, CellSelect, ColumnHeaderSelect, RowHeaderSelect.
- **Ejemplo:**

```
dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
```

15.2 Eventos importantes

a) CellClick

- **Descripción:** Se dispara al hacer clic sobre una celda.
- **Uso práctico:** Permite obtener información de la celda seleccionada o ejecutar acciones según la celda clickeada.
- **Ejemplo:**

```
private void dataGridView1_CellClick(object sender, DataGridViewCellEventArgs e)
{
    MessageBox.Show("Celda seleccionada: " + e.RowIndex + ", " + e.ColumnIndex);
}
```

b) CellValueChanged

- **Descripción:** Se dispara cuando el valor de una celda cambia.
- **Uso práctico:** Ideal para validar datos, calcular totales o actualizar otra parte de la interfaz.
- **Ejemplo:**

```
private void dataGridView1_CellValueChanged(object sender,
DataGridViewCellEventArgs e)
{
    MessageBox.Show("Valor modificado en la fila " + e.RowIndex);
}
```

c) RowHeaderMouseClick

- **Descripción:** Se dispara al hacer clic sobre el encabezado de la fila.
- **Uso práctico:** Permite seleccionar toda la fila o mostrar opciones de contexto.
- **Ejemplo:**

```
private void dataGridView1_RowHeaderMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
    dataGridView1.Rows[e.RowIndex].Selected = true;
}
```

15.3 Métodos importantes

- **Refresh()**
 - Actualiza visualmente el DataGridView.
 - **Ejemplo:** `dataGridView1.Refresh();`
 - **ClearSelection()**
 - Des-selecciona todas las filas y celdas seleccionadas.
 - **Ejemplo:** `dataGridView1.ClearSelection();`
 - **Rows.Add()**
 - Agrega nuevas filas al control.
 - **Ejemplo:** `dataGridView1.Rows.Add("Peras", 0.95);`
 - **Rows.Remove()**
 - Elimina filas específicas.
 - **Ejemplo:** `dataGridView1.Rows.RemoveAt(0);`
-

15.4 Ejemplo práctico: Tabla editable de productos

Supongamos que queremos crear un **inventario simple de productos** donde se pueda agregar, editar y eliminar productos:

```
// Crear DataGridView
DataGridView dataGridView1 = new DataGridView();
dataGridView1.Size = new Size(400, 200);
dataGridView1.Location = new Point(10, 10);
dataGridView1.AllowUserToAddRows = true;
dataGridView1.ReadOnly = false;
dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;

// Crear columnas
dataGridView1.Columns.Add("Producto", "Producto");
dataGridView1.Columns.Add("Precio", "Precio");
dataGridView1.Columns.Add("Cantidad", "Cantidad");

// Agregar filas de ejemplo
dataGridView1.Rows.Add("Manzanas", 1.25, 10);
dataGridView1.Rows.Add("Peras", 0.95, 20);
dataGridView1.Rows.Add("Naranjas", 1.10, 15);
```

```
// Eventos
dataGridView1.CellValueChanged += (s, e) =>
{
    MessageBox.Show("Valor modificado en la fila " + e.RowIndex);
};

dataGridView1.RowHeaderMouseClick += (s, e) =>
{
    dataGridView1.Rows[e.RowIndex].Selected = true;
};

// Agregar DataGridView al formulario
this.Controls.Add(dataGridView1);

// Botón para eliminar fila seleccionada
Button btnEliminar = new Button();
btnEliminar.Text = "Eliminar fila";
btnEliminar.Location = new Point(10, 220);
btnEliminar.Click += (s, e) =>
{
    if (dataGridView1.SelectedRows.Count > 0)
    {
        dataGridView1.Rows.RemoveAt(dataGridView1.SelectedRows[0].Index);
    }
};
this.Controls.Add(btnEliminar);
```

Explicación paso a paso:

1. Se crea un **DataGridView** editable y con selección de fila completa.
2. Se definen tres columnas: Producto, Precio y Cantidad.
3. Se agregan filas iniciales de ejemplo.
4. Se manejan eventos para detectar cambios en celdas y clics en encabezados de fila.
5. Se agrega un botón que permite eliminar la fila seleccionada dinámicamente.

Beneficios del DataGridView:

- Permite **visualización clara de datos tabulares**.

- Facilita **edición y gestión de información** de manera interactiva.
 - Se puede **integrar con bases de datos o colecciones en memoria** para aplicaciones más complejas.
-

Lección 16. Objeto DateTimePicker y MonthCalendar

Los controles **DateTimePicker** y **MonthCalendar** permiten al usuario **seleccionar fechas de manera sencilla y visual**. Son muy útiles en formularios donde se requiere capturar información de fechas como **cumpleaños, fechas de eventos, reservas o registros**.

Estos controles ofrecen propiedades para controlar el **rango de fechas**, el **formato visual** y permiten detectar cambios mediante eventos, lo que facilita la validación y procesamiento de los datos ingresados.

16.1 Propiedades principales

a) DateTimePicker

- **Value**
 - Devuelve o establece la fecha seleccionada en el control.
 - **Ejemplo:**

```
DateTime fechaSeleccionada = dateTimePicker1.Value;
```
 - **Format**
 - Define cómo se muestra la fecha.
 - Opciones comunes: `DateTimePickerFormat.Short`, `DateTimePickerFormat.Long`, `DateTimePickerFormat.Time`, `DateTimePickerFormat.Custom`.
 - **Ejemplo:**

```
dateTimePicker1.Format = DateTimePickerFormat.Short; // dd/MM/yyyy
```
 - **MinDate**
 - Fecha mínima que el usuario puede seleccionar.
 - **Ejemplo:**

```
dateTimePicker1.MinDate = new DateTime(2020, 1, 1);
```
 - **MaxDate**
 - Fecha máxima que el usuario puede seleccionar.
 - **Ejemplo:**

```
dateTimePicker1.MaxDate = new DateTime(2030, 12, 31);
```
-

b) MonthCalendar

- **SelectionStart** y **SelectionEnd**

- Permiten seleccionar un rango de fechas.

- **Ejemplo:**

```
DateTime fechaInicio = monthCalendar1.SelectionStart;
```

```
DateTime fechaFin = monthCalendar1.SelectionEnd;
```

- **MaxSelectionCount**

- Número máximo de días que se pueden seleccionar simultáneamente.

- **Ejemplo:**

```
monthCalendar1.MaxSelectionCount = 7; // máximo una semana
```

- **TodayDate**

- Fecha actual que se resalta en el calendario.

- **Ejemplo:**

```
monthCalendar1.TodayDate = DateTime.Today;
```

16.2 Eventos importantes

ValueChanged (DateTimePicker)

- **Descripción:** Se dispara cuando el usuario cambia la fecha seleccionada.
- **Uso práctico:** Permite actualizar otros controles o realizar validaciones.
- **Ejemplo:**

```
private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
{
    labelFecha.Text = "Fecha seleccionada: " +
dateTimePicker1.Value.ToShortDateString();
}
```

DateChanged y DateSelected (MonthCalendar)

- **DateChanged:** Se dispara cada vez que se cambia la selección de fecha(s).
- **DateSelected:** Se dispara cuando el usuario confirma la selección de fecha(s).
- **Ejemplo:**

```
private void monthCalendar1_DateSelected(object sender, DateRangeEventArgs e)
{
    labelRango.Text = $"Rango seleccionado: {e.Start.ToShortDateString()} -
{e.End.ToShortDateString()}";
}
```

16.3 Métodos importantes

- **Refresh()**
 - Fuerza la actualización visual del control.
 - **Ejemplo:** `dateTimePicker1.Refresh();`
 - **Focus()**
 - Establece el foco en el control para interacción inmediata.
 - **Ejemplo:** `monthCalendar1.Focus();`
-

16.4 Ejemplo práctico: Selección de fechas en formularios

Supongamos que queremos un formulario para **reservar un evento**, donde el usuario puede elegir una fecha específica con `DateTimePicker` y un rango de fechas con `MonthCalendar`:

```
// Crear DateTimePicker
DateTimePicker dateTimePicker1 = new DateTimePicker();
dateTimePicker1.Location = new Point(10, 10);
dateTimePicker1.Format = DateTimePickerFormat.Short;
dateTimePicker1.MinDate = DateTime.Today;
dateTimePicker1.MaxDate = DateTime.Today.AddYears(1);
dateTimePicker1.ValueChanged += (s, e) =>
{
    labelFecha.Text = "Fecha seleccionada: " +
dateTimePicker1.Value.ToShortDateString();
};
this.Controls.Add(dateTimePicker1);

// Crear MonthCalendar
MonthCalendar monthCalendar1 = new MonthCalendar();
monthCalendar1.Location = new Point(10, 50);
monthCalendar1.MaxSelectionCount = 7; // permite seleccionar hasta una semana
monthCalendar1.DateSelected += (s, e) =>
{
    labelRango.Text = $"Rango seleccionado: {e.Start.ToShortDateString()} -
{e.End.ToShortDateString()}";
};
this.Controls.Add(monthCalendar1);
```

```
// Etiquetas para mostrar resultados
Label labelFecha = new Label();
labelFecha.Location = new Point(250, 10);
labelFecha.Size = new Size(200, 20);
this.Controls.Add(labelFecha);
```

```
Label labelRango = new Label();
labelRango.Location = new Point(250, 50);
labelRango.Size = new Size(250, 40);
this.Controls.Add(labelRango);
```

Explicación paso a paso:

1. Se crea un `DateTimePicker` que permite seleccionar una fecha individual dentro de un rango definido.
2. Se maneja el evento `ValueChanged` para actualizar un `Label` con la fecha seleccionada.
3. Se crea un `MonthCalendar` que permite seleccionar un **rango de fechas**, limitado a una semana.
4. Se maneja el evento `DateSelected` para mostrar el rango seleccionado en otra etiqueta.
5. Al combinar ambos controles, se ofrece **flexibilidad en la selección de fechas** para diferentes escenarios: fechas únicas o rangos de eventos.

Beneficios de `DateTimePicker` y `MonthCalendar`

- Facilitan la **entrada de fechas correcta** por parte del usuario.
 - Reducen errores de formato de fecha.
 - Permiten **selección de rangos o fechas individuales** según la necesidad.
 - Se integran fácilmente con otros controles y formularios de datos.
-

Lección 17. Objeto ProgressBar

El **ProgressBar** es un control visual que permite al usuario **ver el progreso de un proceso en ejecución** de manera gráfica. Es ampliamente utilizado en aplicaciones donde se realizan **operaciones que toman tiempo**, como cargas de archivos, descargas, instalaciones o cálculos complejos.

El uso adecuado del ProgressBar mejora la **experiencia de usuario**, ya que proporciona **retroalimentación visual** del estado de la operación.

17.1 Propiedades principales

a) Minimum

- **Descripción:** Valor mínimo que representa el inicio del progreso.
- **Valor por defecto:** 0.
- **Ejemplo:**

```
progressBar1.Minimum = 0;
```

b) Maximum

- **Descripción:** Valor máximo que representa el final del progreso.
- **Uso práctico:** Determina la escala del ProgressBar.
- **Ejemplo:**

```
progressBar1.Maximum = 100;
```

c) Value

- **Descripción:** Valor actual del progreso.
- **Uso práctico:** Permite actualizar manualmente el progreso según el estado del proceso.
- **Ejemplo:**

```
progressBar1.Value = 50; // 50% de progreso
```

d) Step

- **Descripción:** Valor que define cuánto se incrementa el progreso al llamar a `PerformStep()`.
- **Ejemplo:**

```
progressBar1.Step = 10;
```

e) Style

- **Descripción:** Define el estilo visual del ProgressBar.
- **Opciones:**

- Blocks (por defecto, bloques llenando la barra)
- Continuous (barra continua)
- Marquee (animación infinita para procesos sin duración conocida)

- **Ejemplo:**

```
progressBar1.Style = ProgressBarStyle.Continuous;
```

17.2 Métodos importantes

a) PerformStep()

- **Descripción:** Incrementa el valor actual del ProgressBar según la propiedad Step.
- **Ejemplo:**

```
progressBar1.PerformStep();
```

b) Increment()

- **Descripción:** Incrementa el valor actual en un valor específico.
- **Ejemplo:**

```
progressBar1.Increment(5); // Incrementa 5 unidades
```

17.3 Ejemplo práctico: Indicador de carga de procesos

Supongamos que queremos simular un **proceso de carga de archivos** y mostrar el progreso al usuario:

```
// Crear ProgressBar
ProgressBar progressBar1 = new ProgressBar();
progressBar1.Location = new Point(10, 10);
progressBar1.Size = new Size(300, 25);
progressBar1.Minimum = 0;
progressBar1.Maximum = 100;
progressBar1.Step = 10;
progressBar1.Style = ProgressBarStyle.Continuous;
this.Controls.Add(progressBar1);

// Crear botón para iniciar carga
Button btnCargar = new Button();
btnCargar.Text = "Iniciar Carga";
btnCargar.Location = new Point(10, 50);
btnCargar.Click += (s, e) =>
```

```

{
    progressBar1.Value = 0; // Reiniciar progreso

    for (int i = 0; i <= 10; i++)
    {
        System.Threading.Thread.Sleep(200); // Simular proceso
        progressBar1.PerformStep(); // Avanza según Step
        progressBar1.Refresh(); // Refrescar visualización
    }

    MessageBox.Show("Carga completada.");
};
this.Controls.Add(btnCargar);

```

Explicación paso a paso:

1. Se crea un `ProgressBar` con un rango de 0 a 100 y un incremento (`Step`) de 10 unidades.
2. Se agrega un botón que simula un proceso de carga mediante un bucle.
3. Dentro del bucle, se utiliza `Thread.Sleep()` para simular tiempo de proceso, y `PerformStep()` para avanzar la barra.
4. Se llama a `Refresh()` para actualizar la interfaz durante el bucle.
5. Al finalizar, se muestra un mensaje indicando que la carga ha concluido.

Recomendaciones de uso

- Para **procesos de duración conocida**, usar `Minimum`, `Maximum` y `Value` con `PerformStep()` o `Increment()`.
 - Para **procesos de duración desconocida**, utilizar `Style = Marquee` para mostrar una animación continua.
 - Evitar operaciones largas en el hilo principal; usar `BackgroundWorker` o `Task` para mantener la interfaz responsiva.
 - Combinar el `ProgressBar` con etiquetas (`Label`) para mostrar **porcentaje de avance** o detalles del proceso.
-

Lección 18. Objeto TrackBar

El **TrackBar** es un control deslizante que permite al usuario **seleccionar un valor numérico dentro de un rango determinado** mediante el movimiento de un control de “thumb” (manija deslizante).

Es muy útil en interfaces donde se desea **ajustar valores de manera visual e intuitiva**, como el volumen de audio, brillo de pantalla, velocidad de reproducción o cualquier parámetro numérico continuo.

18.1 Propiedades principales

a) Minimum

- **Descripción:** Valor mínimo que puede seleccionar el usuario.
- **Valor por defecto:** 0.
- **Ejemplo:**

```
trackBar1.Minimum = 0;
```

b) Maximum

- **Descripción:** Valor máximo que puede seleccionar el usuario.
- **Ejemplo:**

```
trackBar1.Maximum = 100;
```

c) Value

- **Descripción:** Valor actual del TrackBar.
- **Uso práctico:** Permite obtener o establecer el valor seleccionado.
- **Ejemplo:**

```
trackBar1.Value = 50; // Valor inicial
```

d) TickFrequency

- **Descripción:** Define la frecuencia de las marcas (ticks) visibles en el TrackBar.
- **Uso práctico:** Ayuda al usuario a seleccionar valores discretos más fácilmente.
- **Ejemplo:**

```
trackBar1.TickFrequency = 10; // Muestra un tick cada 10 unidades
```

18.2 Eventos importantes

a) Scroll

- **Descripción:** Se dispara cada vez que el usuario mueve la manija del TrackBar.

- **Uso práctico:** Permite actualizar en tiempo real otros controles según el valor seleccionado.
- **Ejemplo:**

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    labelValor.Text = "Volumen: " + trackBar1.Value + "%";
}
```

b) ValueChanged

- **Descripción:** Se dispara cuando cambia el valor del TrackBar, ya sea por usuario o mediante programación.
- **Uso práctico:** Ideal para reaccionar a cambios de valor y sincronizar otros elementos.
- **Ejemplo:**

```
trackBar1.ValueChanged += (s, e) =>
{
    progressBar1.Value = trackBar1.Value; // Sincroniza con otro control
};
```

18.3 Ejemplo práctico: Control de volumen o brillo

Vamos a crear un **control de volumen** donde el TrackBar permite ajustar el volumen y mostrar el valor en un Label:

```
// Crear TrackBar
TrackBar trackBar1 = new TrackBar();
trackBar1.Location = new Point(10, 10);
trackBar1.Size = new Size(300, 45);
trackBar1.Minimum = 0;
trackBar1.Maximum = 100;
trackBar1.Value = 50;
trackBar1.TickFrequency = 10;
trackBar1.Scroll += (s, e) =>
{
    labelVolumen.Text = "Volumen: " + trackBar1.Value + "%";
};

this.Controls.Add(trackBar1);

// Etiqueta para mostrar valor
Label labelVolumen = new Label();
```

```
labelVolumen.Location = new Point(320, 15);
labelVolumen.Size = new Size(100, 20);
labelVolumen.Text = "Volumen: 50%";
this.Controls.Add(labelVolumen);

// Ejemplo adicional: sincronizar con ProgressBar
ProgressBar progressBar1 = new ProgressBar();
progressBar1.Location = new Point(10, 60);
progressBar1.Size = new Size(300, 25);
progressBar1.Minimum = 0;
progressBar1.Maximum = 100;
progressBar1.Value = trackBar1.Value;
this.Controls.Add(progressBar1);

// Actualizar ProgressBar cuando TrackBar cambie
trackBar1.ValueChanged += (s, e) =>
{
    progressBar1.Value = trackBar1.Value;
};
```

Explicación paso a paso:

1. Se crea un `TrackBar` con valores de 0 a 100 y ticks cada 10 unidades para seleccionar el volumen.
2. Se agrega un `Label` que muestra el valor actual en porcentaje.
3. Se agrega un `ProgressBar` para reflejar visualmente el nivel de volumen sincronizado con el `TrackBar`.
4. Se manejan los eventos `Scroll` y `ValueChanged` para actualizar tanto el `Label` como el `ProgressBar` en tiempo real.

Beneficios del `TrackBar`

- Permite **ajustar valores numéricos de forma rápida e intuitiva**.
 - Proporciona **retroalimentación visual inmediata** al usuario.
 - Fácil de **combinar con otros controles** como `Labels`, `ProgressBars` y `PictureBoxes`.
 - Muy útil en **interfaces multimedia o de configuración de parámetros**.
-

Lección 19. Objeto ToolTip

El **ToolTip** es un control no visual que permite mostrar **información emergente** al usuario cuando pasa el puntero del mouse sobre un control. Este tipo de ayuda contextual mejora la **usabilidad de la aplicación**, facilitando que los usuarios comprendan la función de botones, cuadros de texto, menús u otros elementos de la interfaz.

El ToolTip es **muy útil en interfaces complejas**, donde los controles tienen funciones que podrían no ser evidentes de inmediato.

19.1 Propiedades principales

a) Active

- **Descripción:** Indica si los ToolTips están activos o no.
- **Uso práctico:** Permite habilitar o deshabilitar todas las ayudas emergentes de manera centralizada.
- **Ejemplo:**

```
toolTip1.Active = true; // Activa los ToolTips
```

b) AutomaticDelay

- **Descripción:** Establece el tiempo de retraso automático (en milisegundos) antes de que se muestre un ToolTip.
- **Uso práctico:** Controla la rapidez con la que aparece la información.
- **Ejemplo:**

```
toolTip1AutomaticDelay = 500; // 0.5 segundos
```

c) ToolTipIcon

- **Descripción:** Define el ícono que aparece junto al texto en ToolTips estilo Balloon.
- **Opciones:** None, Info, Warning, Error.
- **Ejemplo:**

```
toolTip1.ToolTipIcon = ToolTipIcon.Info;
```

d) ToolTipTitle

- **Descripción:** Texto que aparece como título en ToolTips estilo Balloon.
- **Uso práctico:** Proporciona un encabezado informativo sobre la descripción emergente.
- **Ejemplo:**

```
toolTip1.ToolTipTitle = "Botón Guardar";
```

19.2 Métodos importantes

a) SetToolTip()

- **Descripción:** Asocia un ToolTip a un control específico con el texto que se mostrará.
- **Ejemplo:**

```
toolTip1.SetToolTip(buttonGuardar, "Guarda los cambios realizados en el formulario");
```

b) RemoveAll()

- **Descripción:** Elimina todos los ToolTips configurados para los controles asociados.
- **Ejemplo:**

```
toolTip1.RemoveAll();
```

19.3 Ejemplo práctico: Descripciones emergentes en botones

Supongamos que queremos un formulario con **tres botones**, cada uno con su ToolTip para explicar su función:

```
// Crear ToolTip
ToolTip toolTip1 = new ToolTip();
toolTip1.Active = true;
toolTip1.AutomaticDelay = 500;
toolTip1.ToolTipIcon = ToolTipIcon.Info;

// Crear botones
Button buttonGuardar = new Button();
buttonGuardar.Text = "Guardar";
buttonGuardar.Location = new Point(10, 10);
this.Controls.Add(buttonGuardar);

Button buttonCancelar = new Button();
buttonCancelar.Text = "Cancelar";
buttonCancelar.Location = new Point(110, 10);
this.Controls.Add(buttonCancelar);

Button buttonEliminar = new Button();
buttonEliminar.Text = "Eliminar";
buttonEliminar.Location = new Point(210, 10);
this.Controls.Add(buttonEliminar);
```

```
// Asignar ToolTips a botones

toolTip1.SetToolTip(buttonGuardar, "Guarda los cambios realizados en el formulario");

toolTip1.SetToolTip(buttonCancelar, "Cancela las modificaciones y cierra el formulario");

toolTip1.SetToolTip(buttonEliminar, "Elimina el registro seleccionado");


// Ejemplo adicional: ToolTip estilo Balloon con título
toolTip1.ToolTipTitle = "Atención";
toolTip1.SetToolTip(buttonEliminar, "Esta acción eliminará el elemento permanentemente");
```

Explicación paso a paso:

1. Se crea un `ToolTip` y se configuran propiedades básicas como `Active`, `AutomaticDelay` y `ToolTipIcon`.
2. Se crean tres botones con distintas funciones en el formulario.
3. Se usa `SetToolTip()` para asociar un texto descriptivo a cada botón, que aparecerá al pasar el mouse sobre ellos.
4. Se puede usar `ToolTipTitle` y `ToolTipIcon` para mostrar información adicional en estilo Balloon.

Beneficios del ToolTip

- Mejora la **usabilidad y accesibilidad** de la aplicación.
 - Proporciona **ayuda contextual** sin ocupar espacio permanente en la interfaz.
 - Facilita que los usuarios comprendan **acciones complejas** o funciones de controles no evidentes.
 - Puede combinarse con **Balloon Tips y títulos** para enfatizar alertas o instrucciones importantes.
-

Quinta Parte: Diálogos y Utilidades del Sistema

En Windows Forms, los **diálogos comunes** permiten al usuario interactuar con el sistema para **abrir archivos, guardar información, seleccionar colores o fuentes**, facilitando la creación de aplicaciones más profesionales y fáciles de usar.

Estos diálogos son controles no visuales que **se muestran de manera modal** y retornan información según la interacción del usuario.

Lección 20. Objetos de Diálogo Común

20.1 OpenFileDialog

El **OpenFileDialog** permite al usuario seleccionar **uno o varios archivos** existentes para abrirlos en la aplicación.

Propiedades principales:

- **FileName:** Nombre del archivo seleccionado.
- **Filter:** Filtro de tipos de archivo a mostrar (por ejemplo, "Archivos de texto|.txt|Todos los archivos|.*").
- **Multiselect:** Permite seleccionar más de un archivo.

Método principal:

- **ShowDialog():** Muestra el diálogo y retorna `DialogResult.OK` si el usuario selecciona un archivo y confirma.

Ejemplo:

```
OpenFileDialog openFileDialog1 = new OpenFileDialog();
openFileDialog1.Filter = "Archivos de texto|.txt|Todos los archivos|.*";
openFileDialog1.Multiselect = true;

if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
    foreach (string archivo in openFileDialog1.FileNames)
    {
        MessageBox.Show("Archivo seleccionado: " + archivo);
    }
}
```

20.2 SaveFileDialog

El **SaveFileDialog** permite al usuario **guardar información en un archivo nuevo** o sobrescribir uno existente.

Propiedades principales:

- **FileName:** Nombre del archivo a guardar.
- **Filter:** Filtro de tipos de archivo.

Método principal:

- **ShowDialog():** Muestra el diálogo y retorna `DialogResult.OK` si el usuario confirma el guardado.

Ejemplo:

```
SaveFileDialog saveFileDialog1 = new SaveFileDialog();
saveFileDialog1.Filter = "Archivos de texto|*.txt|Todos los archivos|*.*";

if (saveFileDialog1.ShowDialog() == DialogResult.OK)
{
    string ruta = saveFileDialog1.FileName;
    File.WriteAllText(ruta, "Contenido de ejemplo");
    MessageBox.Show("Archivo guardado en: " + ruta);
}
```

20.3 FontDialog y ColorDialog

Estos diálogos permiten al usuario **seleccionar fuentes y colores** para personalizar la apariencia de controles, textos o gráficos.

FontDialog

- **Propiedad Font:** Fuente seleccionada por el usuario.
- **Método ShowDialog():** Muestra el diálogo.

Ejemplo:

```
FontDialog fontDialog1 = new FontDialog();
if (fontDialog1.ShowDialog() == DialogResult.OK)
{
    textBox1.Font = fontDialog1.Font;
}
```

ColorDialog

- **Propiedad Color:** Color seleccionado por el usuario.

- **Método ShowDialog():** Muestra el diálogo.

Ejemplo:

```
ColorDialog colorDialog1 = new ColorDialog();
if (colorDialog1.ShowDialog() == DialogResult.OK)
{
    textBox1.ForeColor = colorDialog1.Color;
}
```

20.4 Ejemplo práctico: Editor de texto con cambio de fuente y color

Supongamos que queremos crear un **editor de texto simple** con funcionalidades de abrir, guardar, cambiar fuente y color de texto:

```
// Crear controles
TextBox textBox1 = new TextBox();
textBox1.Multiline = true;
textBox1.Dock = DockStyle.Fill;
this.Controls.Add(textBox1);

// Botón Abrir
Button btnAbrir = new Button();
btnAbrir.Text = "Abrir";
btnAbrir.Location = new Point(10, 10);
btnAbrir.Click += (s, e) =>
{
    OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.Filter = "Archivos de texto|*.txt|Todos los archivos|*.*";
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = File.ReadAllText(openFileDialog1.FileName);
    }
};
this.Controls.Add(btnAbrir);

// Botón Guardar
Button btnGuardar = new Button();
btnGuardar.Text = "Guardar";
```

```

btnGuardar.Location = new Point(100, 10);
btnGuardar.Click += (s, e) =>
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    saveFileDialog1.Filter = "Archivos de texto|*.txt|Todos los archivos|*.*";
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        File.WriteAllText(saveFileDialog1.FileName, textBox1.Text);
    }
};
this.Controls.Add(btnGuardar);

// Botón Fuente
Button btnFuente = new Button();
btnFuente.Text = "Fuente";
btnFuente.Location = new Point(200, 10);
btnFuente.Click += (s, e) =>
{
    FontDialog fontDialog1 = new FontDialog();
    if (fontDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.Font = fontDialog1.Font;
    }
};
this.Controls.Add(btnFuente);

// Botón Color
Button btnColor = new Button();
btnColor.Text = "Color";
btnColor.Location = new Point(300, 10);
btnColor.Click += (s, e) =>
{
    ColorDialog colorDialog1 = new ColorDialog();
    if (colorDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.ForeColor = colorDialog1.Color;
    }
};

```

```
}  
};  
this.Controls.Add(btnColor);
```

Explicación paso a paso:

1. Se crea un `TextBox` multilinea para escribir o mostrar texto.
2. Botón **Abrir** usa `OpenFileDialog` para cargar archivos de texto.
3. Botón **Guardar** usa `SaveFileDialog` para almacenar el contenido en un archivo.
4. Botón **Fuente** permite cambiar la fuente mediante `FontDialog`.
5. Botón **Color** permite cambiar el color del texto mediante `ColorDialog`.
6. La interfaz combina **diálogos del sistema** con controles visuales para crear un **editor funcional y personalizable**.

Beneficios del uso de diálogos comunes

- Facilitan la **interacción del usuario con el sistema de archivos**.
 - Permiten **personalizar la apariencia de la aplicación** mediante fuentes y colores.
 - Proporcionan **herramientas estándar y confiables**, evitando programar funcionalidad compleja desde cero.
 - Mejoran la **experiencia de usuario** al usar controles conocidos y consistentes con Windows.
-

Sexta Parte: Eventos y Funciones Generales

Lección 21. Eventos Globales en Windows Forms

En **Windows Forms**, un **evento** es una acción o suceso que ocurre en un control o en la ventana principal (form), que puede ser detectado y manejado por el programa mediante **manejadores de eventos** (event handlers). Los **eventos globales** son aquellos que pueden afectar a todo el formulario o a múltiples controles de manera centralizada, como los eventos del teclado, del ratón o de enfoque.

Estos eventos son fundamentales para crear interfaces **interactivas y dinámicas**, ya que permiten responder a acciones del usuario en tiempo real.

21.1. Eventos del teclado

Los **eventos del teclado** permiten detectar cuándo el usuario presiona, mantiene o libera una tecla. En Windows Forms, los principales eventos del teclado son:

1. **KeyDown**

- Ocurre cuando el usuario presiona cualquier tecla.
- Permite capturar **teclas especiales** (como Ctrl, Shift, Alt) y combinaciones.

- Sintaxis de suscripción:

```
this.KeyDown += new KeyEventHandler(Form1_KeyDown);
```

- Ejemplo de uso:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        MessageBox.Show("Has presionado Enter");
    }
}
```

2. **KeyPress**

- Ocurre cuando se presiona una tecla que genera un carácter.
- Es útil para **validar entradas de texto**, ya que solo captura caracteres imprimibles.
- Sintaxis:

```
this.KeyPress += new KeyPressEventHandler(Form1_KeyPress);
```

- Ejemplo de uso:

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
```

```

        if (!char.IsDigit(e.KeyChar))
        {
            e.Handled = true; // bloquea cualquier carácter que no sea
número
        }
    }
}

```

3. KeyUp

- Se dispara cuando el usuario **libera una tecla**.
- Es útil para detectar combinaciones de teclas y para controlar acciones continuas.
- Ejemplo:

```

private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Space)
    {
        MessageBox.Show("Se soltó la barra espaciadora");
    }
}

```

Tip: Para que los eventos del teclado funcionen en el formulario principal, se debe establecer la propiedad **KeyPreview = true**, de manera que el formulario pueda recibir los eventos antes que los controles.

```
this.KeyPreview = true;
```

21.2. Eventos del ratón

Los **eventos del ratón** permiten interactuar con el cursor y los clics en controles o formularios:

1. MouseClick

- Ocurre cuando se hace clic en un control o formulario.
- Sintaxis:

```
this.MouseClick += new MouseEventHandler(Form1_MouseClick);
```

- Ejemplo:

```

private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        MessageBox.Show("Clic derecho detectado");
    }
}

```

2. **MouseMove**

- Ocurre cada vez que el ratón se mueve sobre el control o formulario.
- Puede usarse para **mostrar coordenadas** o cambiar dinámicamente la interfaz.
- Ejemplo:

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    this.Text = $"Coordenadas: X={e.X}, Y={e.Y}";
}
```

3. **MouseWheel**

- Detecta el uso de la rueda del ratón.
- Permite hacer zoom o desplazamiento dentro de controles.
- Ejemplo:

```
private void Form1_MouseWheel(object sender, MouseEventArgs e)
{
    if (e.Delta > 0)
        MessageBox.Show("Rueda hacia arriba");
    else
        MessageBox.Show("Rueda hacia abajo");
}
```

Tip: Para que el evento MouseWheel funcione, el control debe tener **Focus**, o suscribirse al evento del formulario.

21.3. Eventos de enfoque

Los **eventos de enfoque** controlan cuándo un control obtiene o pierde el foco de entrada. Son útiles para validaciones, cambios de estilo y mejorar la experiencia del usuario.

1. **Enter**

- Se dispara cuando un control recibe el foco.
- Ejemplo:

```
private void textBox1_Enter(object sender, EventArgs e)
{
    textBox1.BackColor = Color.LightYellow;
}
```

2. **Leave**

- Ocurre cuando un control pierde el foco.

- Ejemplo:

```
private void textBox1_Leave(object sender, EventArgs e)
{
    textBox1.BackColor = Color.White;
}
```

3. GotFocus / LostFocus

- Alternativa a Enter/Leave.
- GotFocus ocurre al recibir el foco, LostFocus al perderlo.
- Sintaxis y ejemplo:

```
textBox1.GotFocus += (s, e) => { textBox1.BackColor =
Color.LightBlue; };
textBox1.LostFocus += (s, e) => { textBox1.BackColor =
Color.White; };
```

Tip: Enter/Leave son preferibles para controles complejos, ya que son más seguros que GotFocus/LostFocus.

21.4. Ejemplo: Interfaz interactiva con atajos de teclado

Vamos a crear un formulario con **atajos de teclado**, manejo de ratón y eventos de enfoque:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        this.KeyPreview = true;

        // Eventos de teclado
        this.KeyDown += Form1_KeyDown;
        this.KeyUp += Form1_KeyUp;

        // Eventos de ratón
        this.MouseClick += Form1_MouseClick;
        this.MouseMove += Form1_MouseMove;

        // Eventos de enfoque
        textBox1.Enter += (s, e) => { textBox1.BackColor = Color.LightYellow; };
        textBox1.Leave += (s, e) => { textBox1.BackColor = Color.White; };
```

```

}

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Control && e.KeyCode == Keys.S)
        MessageBox.Show("Atajo Guardar Ctrl+S");
    if (e.KeyCode == Keys.Escape)
        this.Close();
}

private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    // Mostrar tecla liberada
    this.Text = $"Última tecla liberada: {e.KeyCode}";
}

private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
        MessageBox.Show("Clic derecho en el formulario");
}

private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    label1.Text = $"Coordenadas del ratón: X={e.X}, Y={e.Y}";
}
}

```

Características del ejemplo:

- **Atajos de teclado** (Ctrl+S, Escape)
- **Cambio de color en TextBox** al recibir o perder foco
- **Detección de clic derecho y movimiento del ratón**
- **Feedback visual** de las coordenadas del cursor

Este ejemplo muestra cómo combinar **eventos globales** para crear una interfaz interactiva y dinámica, mejorando la **experiencia de usuario**.

Resumen

- **Eventos globales** permiten que el formulario y los controles respondan a acciones del usuario.
 - **Eventos de teclado:** KeyDown, KeyPress, KeyUp
 - **Eventos de ratón:** MouseClick, MouseMove, MouseWheel
 - **Eventos de enfoque:** Enter, Leave, GotFocus, LostFocus
 - **Práctica recomendada:** usar **KeyPreview = true** para capturar eventos de teclado en todo el formulario y usar Enter/Leave en lugar de GotFocus/LostFocus para seguridad.
 - Combinar eventos globales permite crear **interfaces más interactivas y responsivas**.
-

Lección 22. Funciones Globales y de Control

En **Windows Forms**, los **controles** (botones, cuadros de texto, etiquetas, etc.) y el **formulario** comparten un conjunto de **métodos comunes** que permiten manipular su comportamiento, apariencia y ciclo de vida. Estas **funciones globales y de control** son esenciales para desarrollar interfaces dinámicas y adaptativas. Además, es posible crear y eliminar controles **dinámicamente en tiempo de ejecución**, lo que permite que la interfaz se ajuste a las necesidades del usuario.

22.1. Métodos comunes de todos los controles

Los controles en Windows Forms heredan de la clase base **Control**, que proporciona varios métodos fundamentales:

1. Focus()

- Da **foco de entrada** a un control específico.
- Útil para guiar al usuario hacia un control en particular.
- Ejemplo:

```
textBox1.Focus(); // Coloca el cursor en textBox1
```

- Buenas prácticas:
 - Solo debe llamarse cuando el control es **visible y habilitado**.
 - Se puede usar dentro de eventos para mejorar la navegación.
-

2. Refresh()

- Fuerza la **actualización inmediata** del control.
- Se utiliza cuando se realizan cambios gráficos y queremos que se reflejen de inmediato.
- Ejemplo:

```
label1.Text = "Cargando...";
```

```
label1.Refresh(); // Actualiza inmediatamente la etiqueta
```

- Diferencia con Invalidate(): Refresh() llama automáticamente a Invalidate() + Paint().
-

3. Invalidate()

- Marca el control para **redibujarse** en el próximo ciclo de actualización.
- No refresca inmediatamente, solo indica que necesita actualización.
- Ejemplo:

```
panel1.Invalidate(); // Solicita que el panel se redibuje
```

- Útil cuando se realizan cambios gráficos complejos en Paint().
-

4. Show() / Hide()

- **Show()**: Hace visible un control oculto.
- **Hide()**: Oculta un control sin eliminarlo.
- Ejemplo:

```
button1.Hide(); // Oculta el botón
```

```
button1.Show(); // Vuelve a mostrarlo
```

- Diferencia: **Visible = true/false** hace lo mismo, pero Show()/Hide() son métodos más intuitivos.
-

5. Dispose()

- Libera los **recursos del sistema** utilizados por un control.
- Se usa para controles que ya no se necesitan y que consumen memoria.
- Ejemplo:

```
button1.Dispose(); // Elimina el botón y libera recursos
```

- Buenas prácticas:
 - Siempre llamar a Dispose() si se crean controles dinámicamente y no se van a usar.
 - Evita **fugas de memoria** en aplicaciones de larga ejecución.
-

22.2. Manejo dinámico de controles en tiempo de ejecución

En Windows Forms, se pueden **crear y eliminar controles desde código** de manera dinámica, lo que permite interfaces más flexibles y adaptables.

Crear controles dinámicamente

- Se instancia un control, se configuran sus propiedades y se agrega al formulario o a un contenedor.
- Ejemplo: crear un botón en tiempo de ejecución

```
Button btn = new Button();
```

```
btn.Text = "Click Dinámico";
```

```
btn.Location = new Point(50, 50);
```

```
btn.Size = new Size(120, 30);
```

```
btn.Click += (s, e) => MessageBox.Show("Botón dinámico presionado");
```

```
this.Controls.Add(btn); // Agrega el botón al formulario
```

Eliminar controles dinámicamente

- Se pueden eliminar controles usando **Controls.Remove()** y luego **Dispose()**.
- Ejemplo:

```
this.Controls.Remove(btn);  
btn.Dispose();
```

- Nota: Siempre es recomendable llamar a **Dispose()** para liberar memoria.

Listas de controles

- Es útil guardar referencias a controles dinámicos en **listas** para manipularlos fácilmente.

```
List<Button> botones = new List<Button>();  
botones.Add(btn);
```

22.3. Ejemplo: Formulario que genera botones dinámicos

Vamos a crear un formulario que genera **botones dinámicos** según un número indicado por el usuario:

```
public partial class Form1 : Form  
{  
    List<Button> botones = new List<Button>();  
  
    public Form1()  
    {  
        InitializeComponent();  
        buttonCrear.Click += ButtonCrear_Click;  
        buttonEliminar.Click += ButtonEliminar_Click;  
    }  
  
    private void ButtonCrear_Click(object sender, EventArgs e)  
    {  
        int cantidad;  
        if (int.TryParse(textBoxCantidad.Text, out cantidad))  
        {  
            int x = 20, y = 60;  
            for (int i = 0; i < cantidad; i++)  
            {  
                Button btn = new Button();  
                btn.Text = $"Botón {i + 1}";  
                btn.Size = new Size(100, 30);  
                btn.Location = new Point(x, y);  
                btn.Click += Btn_Click;  
                this.Controls.Add(btn);  
            }  
        }  
    }  
}
```

```

        botones.Add(btn);

        y += 40; // Ajusta la posición vertical
    }
}
else
{
    MessageBox.Show("Introduce un número válido");
}
}

private void Btn_Click(object sender, EventArgs e)
{
    Button btn = sender as Button;
    MessageBox.Show($"Has presionado {btn.Text}");
}

private void ButtonEliminar_Click(object sender, EventArgs e)
{
    foreach (var btn in botones)
    {
        this.Controls.Remove(btn);
        btn.Dispose();
    }
    botones.Clear();
}
}

```

Explicación del ejemplo:

1. El usuario indica la cantidad de botones que desea generar.
2. Se crean botones dinámicamente, se configuran propiedades y se agregan al formulario.
3. Cada botón tiene un **evento Click** dinámico que muestra su nombre.
4. Un botón “Eliminar” limpia todos los botones creados, usando **Remove()** y **Dispose()** para liberar recursos.
5. Se utiliza una **lista** para mantener referencias y poder manipular los botones fácilmente.

Buenas prácticas:

- Guardar referencias a controles dinámicos.
 - Liberar recursos con **Dispose()**.
 - Evitar posiciones fijas cuando se generan muchos controles; usar contenedores como **FlowLayoutPanel** o **TableLayoutPanel** para organizar automáticamente.
-

Resumen

- Los **métodos comunes** de todos los controles: **Focus()**, **Refresh()**, **Invalidate()**, **Show()**, **Hide()**, **Dispose()** permiten manejar la apariencia, comportamiento y ciclo de vida de los controles.
 - El **manejo dinámico de controles** permite crear interfaces flexibles y adaptables según las necesidades del usuario.
 - **Crear y eliminar controles** correctamente requiere administrar eventos, posiciones y memoria.
 - Las **listas de controles** facilitan su manipulación y liberación de recursos.
 - La combinación de estos métodos con eventos globales permite crear aplicaciones **interactivas, dinámicas y eficientes**.
-

Séptima Parte: Proyectos Prácticos con Objetos y Eventos

En esta sección aplicaremos todo lo aprendido sobre **controles, eventos y funciones globales** para crear proyectos completos en **Windows Forms**. Cada proyecto combina **objetos, eventos y manejo dinámico** de controles para desarrollar aplicaciones prácticas. Estos ejemplos sirven como base para proyectos reales y más complejos.

23. Proyecto 1: Calculadora Científica con TextBox y Buttons

Objetivo

Crear una **calculadora científica** que permita operaciones básicas y funciones matemáticas avanzadas usando **TextBox** y **Buttons**.

Controles utilizados

- **TextBox**: para mostrar y capturar números y resultados.
- **Buttons**: para números, operadores y funciones (suma, resta, sin, cos, log, etc.).
- **Eventos**: **Click** de cada botón, **KeyPress** para validar entrada.

Estructura y lógica

1. Captura de números y operadores desde botones o teclado.
2. Almacenar el número actual y operador seleccionado.
3. Ejecutar la operación al presionar "=".
4. Funciones científicas (sqrt, sin, cos, log, etc.) aplicadas al número actual.

Ejemplo de implementación

```
public partial class FormCalculadora : Form
{
    double valorActual = 0;
    string operador = "";
    bool nuevaOperacion = true;

    public FormCalculadora()
    {
        InitializeComponent();
        foreach (Control c in this.Controls)
        {
            if (c is Button)
```

```

        {
            c.Click += Button_Click;
        }
    }
}

```

```

private void Button_Click(object sender, EventArgs e)
{
    Button btn = sender as Button;
    if (char.IsDigit(btn.Text, 0) || btn.Text == ".")
    {
        if (nuevaOperacion)
        {
            textBox1.Text = btn.Text;
            nuevaOperacion = false;
        }
        else
        {
            textBox1.Text += btn.Text;
        }
    }
    else
    {
        switch (btn.Text)
        {
            case "+": case "-": case "*": case "/":
                operador = btn.Text;
                valorActual = double.Parse(textBox1.Text);
                nuevaOperacion = true;
                break;
            case "=":
                EjecutarOperacion();
                break;
            case "C":
                textBox1.Text = "0";
                valorActual = 0;

```

```

        operador = "";
        nuevaOperacion = true;
        break;
    case "sin":
        textBox1.Text =
Math.Sin(double.Parse(textBox1.Text)).ToString();
        nuevaOperacion = true;
        break;
        // Agregar más funciones científicas
    }
}

private void EjecutarOperacion()
{
    double valorNuevo = double.Parse(textBox1.Text);
    switch (operador)
    {
        case "+": textBox1.Text = (valorActual + valorNuevo).ToString();
break;
        case "-": textBox1.Text = (valorActual - valorNuevo).ToString();
break;
        case "*": textBox1.Text = (valorActual * valorNuevo).ToString();
break;
        case "/": textBox1.Text = (valorActual / valorNuevo).ToString();
break;
    }
    nuevaOperacion = true;
}
}

```

Claves del proyecto:

- Uso intensivo de **eventos Click**.
 - Validación de entrada con **KeyPress**.
 - Manejo de **estado de la operación** (valorActual, operador, nuevaOperacion).
-

24. Proyecto 2: Agenda de Contactos con ListBox y Archivos

Objetivo

Crear una **agenda de contactos** que permita agregar, eliminar, buscar y almacenar contactos en un archivo.

Controles utilizados

- TextBox para capturar nombre y teléfono.
- ListBox para mostrar la lista de contactos.
- Button para agregar, eliminar, guardar y cargar contactos.
- Archivos: StreamWriter y StreamReader.

Estructura y lógica

1. Guardar contactos en **lista dinámica**.
2. Mostrar contactos en **ListBox**.
3. Guardar y cargar contactos desde un archivo de texto.
4. Permitir búsqueda y eliminación de contactos.

Ejemplo de implementación

```
public partial class FormAgenda : Form
{
    List<string> contactos = new List<string>();
    string archivo = "contactos.txt";

    public FormAgenda()
    {
        InitializeComponent();
        buttonAgregar.Click += ButtonAgregar_Click;
        buttonEliminar.Click += ButtonEliminar_Click;
        buttonGuardar.Click += ButtonGuardar_Click;
        buttonCargar.Click += ButtonCargar_Click;
    }

    private void ButtonAgregar_Click(object sender, EventArgs e)
    {
        string contacto = $"{textBoxNombre.Text} - {textBoxTelefono.Text}";
        contactos.Add(contacto);
    }
}
```

```

        listBox1.Items.Add(contacto);
        textBoxNombre.Clear(); textBoxTelefono.Clear();
    }

private void ButtonEliminar_Click(object sender, EventArgs e)
{
    if (listBox1.SelectedIndex >= 0)
    {
        contactos.RemoveAt(listBox1.SelectedIndex);
        listBox1.Items.RemoveAt(listBox1.SelectedIndex);
    }
}

private void ButtonGuardar_Click(object sender, EventArgs e)
{
    using (StreamWriter sw = new StreamWriter(archivo))
    {
        foreach (string c in contactos)
            sw.WriteLine(c);
    }
}

private void ButtonCargar_Click(object sender, EventArgs e)
{
    contactos.Clear();
    listBox1.Items.Clear();
    if (File.Exists(archivo))
    {
        using (StreamReader sr = new StreamReader(archivo))
        {
            string linea;
            while ((linea = sr.ReadLine()) != null)
            {
                contactos.Add(linea);
                listBox1.Items.Add(linea);
            }
        }
    }
}

```

```

        }
    }
}

```

Claves del proyecto:

- Manejo de **ListBox** y **List** simultáneamente.
- Lectura y escritura en archivos de forma segura con `using`.
- Validación simple de selección antes de eliminar.

25. Proyecto 3: Reproductor de Imágenes con PictureBox y Timer

Objetivo

Crear un **reproductor de imágenes** que muestre fotos en secuencia con temporizador y permita avanzar o retroceder manualmente.

Controles utilizados

- `PictureBox` para mostrar imágenes.
- `Button` para controlar inicio, pausa, siguiente y anterior.
- `Timer` para automatizar el pase de imágenes.

Estructura y lógica

1. Cargar imágenes de una carpeta.
2. Mostrar imagen actual en `PictureBox`.
3. Avanzar automáticamente con `Timer`.
4. Controles manuales para cambiar imagen.

Ejemplo de implementación

```

public partial class FormReproductor : Form
{
    List<Image> imagenes = new List<Image>();
    int indice = 0;

    public FormReproductor()
    {
        InitializeComponent();
        timer1.Interval = 2000;
    }
}

```

```

timer1.Tick += Timer1_Tick;
buttonIniciar.Click += (s, e) => timer1.Start();
buttonPausar.Click += (s, e) => timer1.Stop();
buttonSiguiente.Click += ButtonSiguiente_Click;
buttonAnterior.Click += ButtonAnterior_Click;

// Cargar imágenes
string[] archivos = Directory.GetFiles("imagenes", "*.jpg");
foreach (string f in archivos)
    imagenes.Add(Image.FromFile(f));

if (imagenes.Count > 0)
    pictureBox1.Image = imagenes[0];
}

private void Timer1_Tick(object sender, EventArgs e)
{
    indice = (indice + 1) % imagenes.Count;
    pictureBox1.Image = imagenes[indice];
}

private void ButtonSiguiente_Click(object sender, EventArgs e)
{
    indice = (indice + 1) % imagenes.Count;
    pictureBox1.Image = imagenes[indice];
}

private void ButtonAnterior_Click(object sender, EventArgs e)
{
    indice = (indice - 1 + imagenes.Count) % imagenes.Count;
    pictureBox1.Image = imagenes[indice];
}
}

```

Claves del proyecto:

- Uso de **Timer** para automatización.
- Control de índice circular con módulo (%).

- Carga de imágenes desde carpeta externa.

26. Proyecto 4: Sistema de Inventario con DataGridView y Menús

Objetivo

Crear un **sistema de inventario** que permita agregar, editar y eliminar productos, con visualización en **DataGridView** y opciones de menú.

Controles utilizados

- DataGridView para mostrar productos.
- MenuStrip con opciones como Agregar, Editar, Eliminar.
- TextBox, NumericUpDown y ComboBox para capturar datos.

Estructura y lógica

1. Guardar productos en lista de objetos **Producto**.
2. Mostrar productos en **DataGridView**.
3. Menú para gestionar operaciones CRUD.
4. Validaciones de datos antes de agregar o modificar.

Ejemplo de clase Producto

```
public class Producto
{
    public string Nombre { get; set; }
    public int Cantidad { get; set; }
    public double Precio { get; set; }
}
```

Ejemplo de agregar producto

```
List<Producto> inventario = new List<Producto>();

private void agregarToolStripMenuItem_Click(object sender, EventArgs e)
{
    Producto p = new Producto()
    {
        Nombre = textBoxNombre.Text,
        Cantidad = (int)numericUpDownCantidad.Value,
        Precio = double.Parse(textBoxPrecio.Text)
    }
}
```

```

};
inventario.Add(p);
dataGridView1.DataSource = null;
dataGridView1.DataSource = inventario;
}

```

Uso de `DataGridView.DataSource` para refrescar datos.

- Menú para centralizar operaciones.
- Validaciones de entrada antes de operaciones CRUD.

27. Proyecto 5: Editor de Texto con Diálogos de Fuente y Color

Objetivo

Crear un **editor de texto básico** que permita cambiar fuente, tamaño y color del texto usando diálogos estándar.

Controles utilizados

- `RichTextBox` para el área de edición.
- `MenuStrip` para opciones como Fuente, Color, Abrir, Guardar.
- `FontDialog` y `ColorDialog` para selección visual.

Ejemplo de cambiar fuente y color

```

private void fuenteToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (fontDialog1.ShowDialog() == DialogResult.OK)
    {
        richTextBox1.Font = fontDialog1.Font;
    }
}

private void colorToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (colorDialog1.ShowDialog() == DialogResult.OK)
    {
        richTextBox1.ForeColor = colorDialog1.Color;
    }
}

```

Abrir y guardar archivo

```
private void abrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        richTextBox1.LoadFile(openFileDialog1.FileName,
RichTextBoxStreamType.PlainText);
    }
}

private void guardarToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        richTextBox1.SaveFile(saveFileDialog1.FileName,
RichTextBoxStreamType.PlainText);
    }
}
```

Claves del proyecto:

- Integración de **diálogos estándar** para fuente y color.
- Uso de `RichTextBox` para soportar texto enriquecido.
- Operaciones de archivo seguras con `OpenFileDialog` y `SaveFileDialog`.

Resumen de la Séptima Parte

- Cada proyecto combina **objetos, controles y eventos** para crear aplicaciones prácticas.
 - Se aplican métodos y eventos globales, manejo dinámico de controles y validaciones.
 - Los proyectos abarcan desde **calculadoras hasta editores de texto**, mostrando la versatilidad de Windows Forms.
 - Buenas prácticas: manejo de listas de objetos, liberación de recursos, validación de entrada y actualización de interfaces dinámicas.
-

Anexos de Windows Forms

Los **anexos** son herramientas de referencia que permiten al desarrollador acceder rápidamente a información clave sobre controles, eventos, métodos y propiedades de Windows Forms. Estos son indispensables para la creación de aplicaciones eficientes y para repasar conceptos de manera rápida.

A. Tabla completa de controles y sus eventos

En Windows Forms, cada control tiene **eventos específicos** y otros heredados de la clase base **Control**. A continuación se presenta una tabla con los controles más comunes y sus eventos más importantes:

Control	Eventos principales	Descripción
Button	Click, MouseEnter, MouseLeave	Detecta clics y cambios del ratón sobre el botón
TextBox	TextChanged, KeyPress, KeyDown, Enter, Leave	Captura texto, teclas y enfoque
Label	Click, MouseEnter, MouseLeave	Interacción básica, útil para eventos visuales
ComboBox	SelectedIndexChanged, DropDown, TextChanged	Cambio de selección o apertura de lista
ListBox	SelectedIndexChanged, MouseDoubleClick	Selección y doble clic
CheckBox	CheckedChanged	Detecta cambios en estado (marcado/no marcado)
RadioButton	CheckedChanged	Detecta cambio de selección individual
PictureBox	Click, MouseMove, MouseClick	Interacción con imágenes
Panel	Paint, MouseClick, MouseMove	Contenedor que permite eventos gráficos y de interacción
Form	Load, Shown, Closing, KeyDown, KeyUp, MouseClick, Resize	Eventos generales del formulario, teclado y ratón
Timer	Tick	Evento repetitivo según intervalo definido
DataGridView	CellClick, CellValueChanged, RowEnter	Interacción con celdas y filas
MenuStrip	ItemClicked	Detecta selección de opciones de menú
RichTextBox	TextChanged, KeyPress, Enter, Leave	Texto enriquecido, captura de teclas y enfoque

Tip: Los eventos **heredados de Control** (Click, MouseEnter, MouseLeave, Enter, Leave, GotFocus, LostFocus) están disponibles en casi todos los controles y permiten una programación consistente.

B. Tabla de métodos comunes en Windows Forms

Todos los controles heredan de la clase `Control`, lo que les proporciona **métodos globales** que facilitan su manejo:

Método	Descripción	Ejemplo práctico
<code>Focus()</code>	Establece el foco de entrada en el control	<code>textBox1.Focus();</code>
<code>Refresh()</code>	Fuerza actualización inmediata del control	<code>label1.Refresh();</code>
<code>Invalidate()</code>	Marca el control para redibujarse en el siguiente ciclo de actualización	<code>panel1.Invalidate();</code>
<code>Show()</code>	Muestra el control si estaba oculto	<code>button1.Show();</code>
<code>Hide()</code>	Oculto el control sin eliminarlo	<code>button1.Hide();</code>
<code>Dispose()</code>	Libera los recursos del control y lo elimina de la memoria	<code>button1.Dispose();</code>
<code>BringToFront()</code>	Trae el control al frente en su contenedor	<code>panel1.BringToFront();</code>
<code>SendToBack()</code>	Envía el control al fondo de su contenedor	<code>panel1.SendToBack();</code>
<code>Update()</code>	Redibuja el control inmediatamente si ha sido invalidado	<code>panel1.Update();</code>
<code>CreateControl()</code>	Fuerza la creación del control antes de mostrarlo	<code>textBox1.CreateControl();</code>
Tip: <code>Refresh() = Invalidate() + Update()</code> , por lo que <code>Refresh()</code> es más inmediato.		

C. Propiedades más utilizadas por tipo de control

Cada control tiene **propiedades específicas** que determinan su apariencia, comportamiento y estado. Aquí se presentan las más relevantes:

Control	Propiedades más utilizadas	Descripción breve
Button	Text, Enabled, Visible, BackColor, ForeColor, Font	Texto, estado activo, colores y fuente
TextBox	Text, ReadOnly, Multiline, MaxLength, PasswordChar, BackColor	Contenido, restricción de entrada, apariencia
Label	Text, AutoSize, ForeColor, BackColor	Texto, tamaño automático y colores
ComboBox	Items, SelectedIndex, SelectedItem, DropDownStyle, Text	Elementos de la lista y selección
ListBox	Items, SelectedIndex, SelectedItem, SelectionMode	Elementos, selección simple o múltiple
CheckBox	Checked, Text, ThreeState, AutoCheck	Estado marcado y comportamiento de tres estados
RadioButton	Checked, Text, AutoCheck	Estado de selección
PictureBox	Image, SizeMode, BorderStyle, BackColor	Imagen, ajuste, borde y fondo
Panel	BackColor, BorderStyle, AutoScroll, Controls	Color de fondo, borde, scroll y contenedores hijos
Form	Text, Size, StartPosition, FormBorderStyle, KeyPreview, MaximizeBox	Título, tamaño, posición, borde, eventos de teclado
Timer	Interval, Enabled	Intervalo en ms y estado activo

Control	Propiedades más utilizadas	Descripción breve
DataGridView	DataSource, Columns, Rows, ReadOnly, SelectionMode	Fuente de datos, configuración de columnas y filas
MenuStrip	Items, Dock, Enabled	Elementos del menú, anclaje y estado
RichTextBox	Text, Rtf, SelectionFont, SelectionColor, ReadOnly	Texto enriquecido, fuente y color seleccionados, solo lectura

Tip: Propiedades como **Visible**, **Enabled** y **BackColor** se aplican prácticamente a todos los controles, permitiendo un manejo global uniforme.

Resumen de los anexos

1. **Tabla de controles y eventos:** Referencia rápida de los controles más comunes y los eventos más importantes para interactuar con ellos.
2. **Tabla de métodos comunes:** Funciones globales de **Control** que permiten manejar el ciclo de vida, actualización y visibilidad de cualquier control.
3. **Propiedades más utilizadas:** Cada control tiene propiedades específicas, y algunas son globales a todos, permitiendo consistencia y facilidad de manejo.

Estos anexos funcionan como **manual de consulta rápida**, ideal para programadores que trabajan con **Windows Forms** y quieren **acceder a eventos, métodos y propiedades sin tener que buscar en documentación extensa**.

Epílogo

Llegamos al final de este viaje por el mundo de **C# y Windows Forms**, un camino que comenzó con conceptos básicos y ejemplos sencillos, y que nos llevó a desarrollar aplicaciones completas, interactivas y funcionales. Si algo queda claro, es que la programación no es solo escribir código: es aprender a pensar de manera lógica, estructurada y creativa, resolviendo problemas de formas cada vez más eficientes.

Windows Forms nos ha permitido experimentar de manera visual cómo nuestras ideas pueden convertirse en interfaces concretas, con botones, cuadros de texto, menús y gráficos que interactúan con el usuario. Más allá de la teoría, hemos explorado patrones, buenas prácticas y técnicas que te ayudarán a construir aplicaciones robustas, mantenibles y escalables.

Este libro no es un final, sino un punto de partida. Los conocimientos adquiridos aquí te permiten enfrentarte a retos más grandes: integrar bases de datos, manejar eventos complejos, crear controles personalizados, e incluso explorar otras tecnologías de la familia .NET como WPF o UWP. La programación es un aprendizaje continuo, y cada proyecto será una oportunidad para profundizar y mejorar.

Recuerda que el código que escribes es un reflejo de tu pensamiento: claro, organizado y creativo. Mantén la curiosidad, experimenta sin miedo y nunca dejes de explorar nuevas herramientas y enfoques. Cada error es una lección, cada solución es un logro, y cada proyecto terminado es una muestra de tu crecimiento como desarrollador.

Con estas herramientas y conocimientos, estás lista para crear aplicaciones que no solo funcionen, sino que también sean útiles, eficientes y elegantes. Ahora, el siguiente paso está en tus manos: seguir practicando, innovando y llevando tus ideas más allá de la pantalla.